University of Notre Dame
Department of Electrical Engineering

# Anomaly Detection PCB for CubeSat Missions

Ambroise Curutchague, Bridget Goodwine, Jacob Gose, Conor O'Brien, Zach Zarzaur
Senior Design II
Professor Schafer

## ● Table of Contents

## 1 Introduction

CubeSats are miniature, 10x10x10cm satellites intended for operation in low Earth orbit. They may be used for a number of purposes: placing and using research equipment in space, amateur radio communications, or testing systems when their installation on a larger spacecraft is not yet robust enough to justify the cost. Because of their lightweight, small-scale, and relatively inexpensive design, CubeSats are built by university research groups, clubs, and companies around the world.

IrishSat is a student-led organization with the ultimate goal of launching a CubeSat through NASA’s CubeSat Launch Initiative (CSLI) program. In this program, NASA selects groups developing CubeSats based on proposals for the CubeSat’s research and design. Once built, a selected CubeSat will be placed on-board a NASA mission and ejected into Low Earth Orbit. IrishSat is a fairly young club, having only been around for about 3 years, and consists of many majors, but mainly Electrical Engineering, Mechanical Engineering, Aerospace Engineering, and Computer Science majors. In addition to eventually launching a satellite, the club also strives to help educate young engineers, allow them to apply the content they learn in class, expose them to industry practices, and connect them to industry contacts.

IrishSat is composed of 4 main projects (ProtoSat, OAT Lab, IRIS, and Ground Station) and 4 squads (Power, Computing, STOC, and Communications). Club members typically join a squad and subsequently will join a project to work on their specific area of expertise. The club is led by a President, CTO, Director of Research, and Director of Business Operations, and the projects are led by Chief Engineers and Project Managers.

Over the past 3 years IrishSat has developed three versions of its original High Altitude Balloon project, which sought to simulate the operation of a CubeSat-form-factor embedded system in the harsh environment of outer atmosphere. These High Altitude Balloon missions allowed IrishSat personnel to test all of the main systems that comprise a CubeSat, including power, communications, structure, and computing.

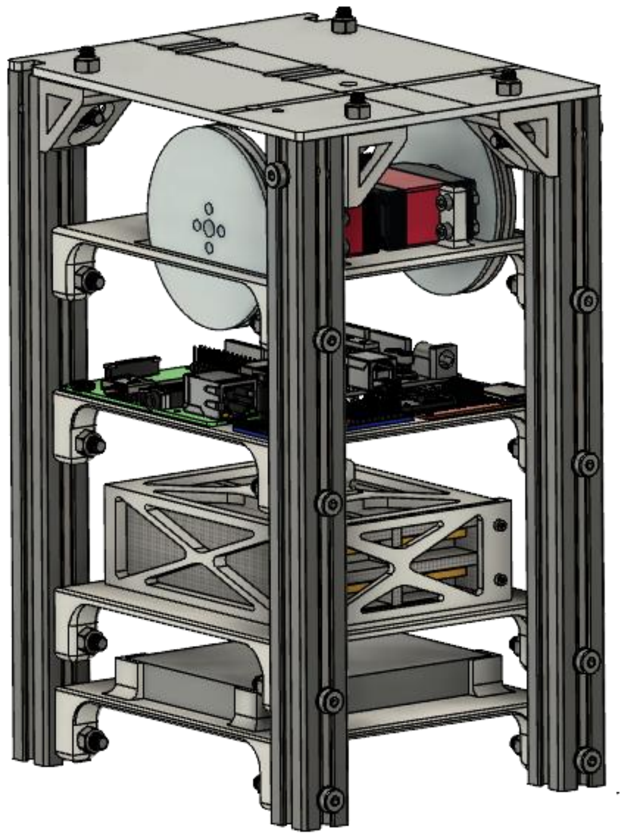


**Figure 1.** CAD drawing of IrishSat's third High Altitude Balloon Project iteration: IRIS v3

In this time period, IrishSat has also developed a Helmholtz cage that acts to simulate an arbitrary magnetic field that can be used to test a CubeSat attitude control system.

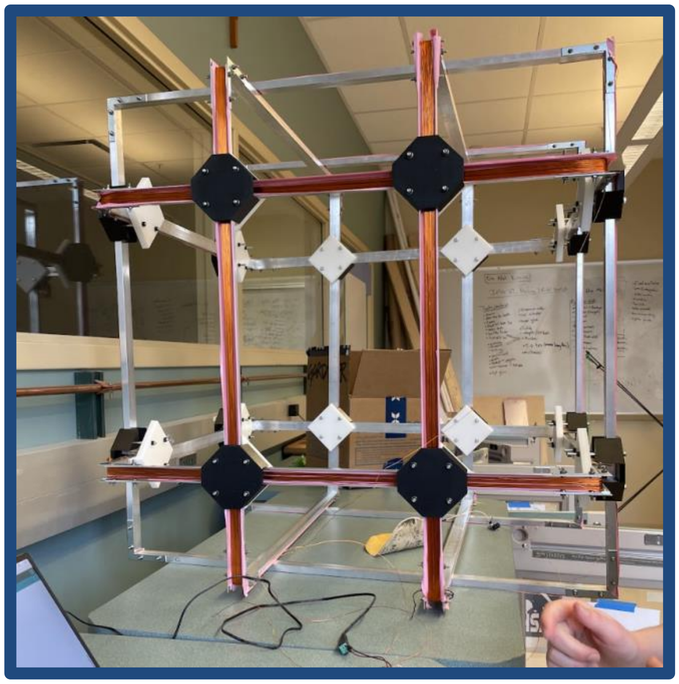


**Figure 2.** Helmholz cage developed by IrishSat's Orbital, Attitude, and Thermal Laboratory

IrishSat has additionally designed, constructed, and tested a Ground Station on the roof of Nieuwland Hall of Science that allows our communications squad to track and communicate with satellites currently in orbit, and will eventually serve as the main data link between our CubeSat and Earth.

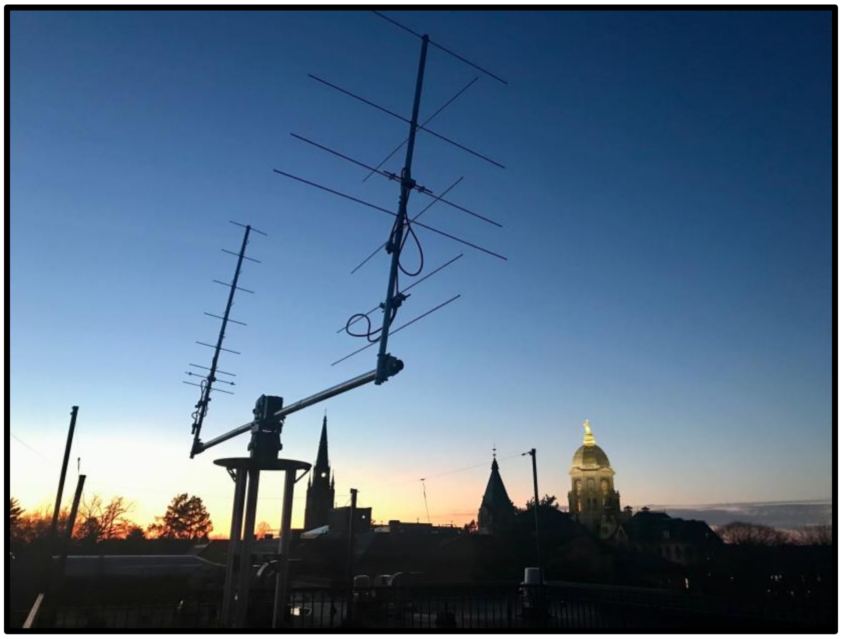


**Figure 3.** Uplink and downlink antenna elements for IrishSat's Ground Station

Finally, and most important to this Senior Design project, IrishSat has also developed two iterations of a prototype CubeSat as part of their Proto-Sat project, testing structure, power, attitude control, computing, and communications. Last year's initial prototype was successfully demonstrated to have a working attitude control system that was able to use reaction wheels to slow an induced rotation. This year's prototype was constructed in a 1U (10x10x10 cm) CubeSat form factor, and was intended to be a more complete prototype, integrating all CubeSat subsystems into one, wholly independently operating package.

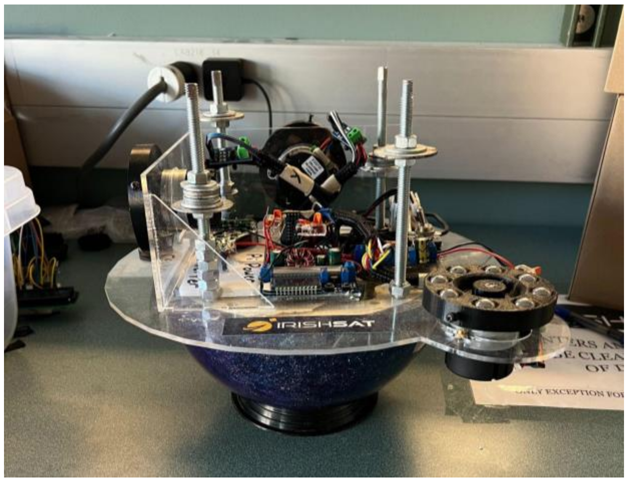


**Figure 4.** ProtoSat's initial prototype

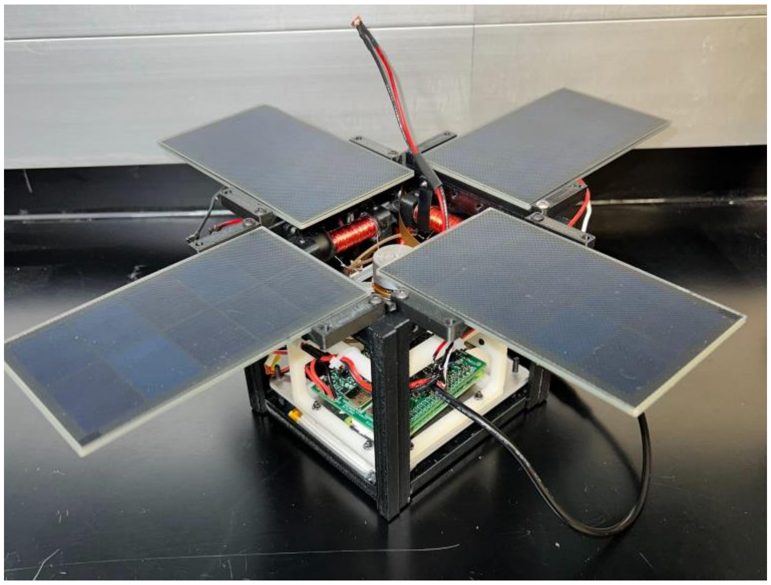


**Figure 5.** ProtoSat's current CubeSat prototype with senior design board integrated (bottom middle)

The goal of this senior design project was to design the “brain” and the “guts”, so to speak, of the prototype CubeSat that IrishSat’s Proto-Sat team has been developing. Since most of the work of the Proto-Sat team has gone into the design of other systems, we decided that it would be a great load off of the Proto-Sat team’s shoulders if the heavy work for board design was done by a separate, dedicated team. In addition, most of the members of Proto-Sat are underclassmen electrical engineers with no prior experience doing board design or the kind of system-level integration that this project required. By handling the difficult work of designing a power system and state/anomaly detection system that would properly bias all components while ensuring that power was consumed in the most efficient way possible, we intended to contribute significant, practical progress to the Proto-Sat project that heretofore had concentrated work solely on things like the attitude control and structural systems. By taking care of this “dirty work”, we ensured that the underclassmen engineers of Proto-Sat, already a personnel-limited team, were free to concentrate on the design of other systems.

It is a well known fact that the main failure point for student-designed CubeSat systems is that they most often run out of power before they even get the opportunity to deploy. After a CubeSat is handed over to NASA, it will sit for days or even weeks before being launched, during which time it is confined to a dark space with no opportunity for batteries to charge. By handling the optimization of power consumption in a robust manner, we are giving our CubeSat a much better chance for success. Implementing a state and anomaly detection system allows for our CubeSat to use power only when it needs to, and additionally, to report back to our ground station when it measures currents and voltages that indicate components going haywire. Our framework allows for these issues to be detected and remedied, whether through automatic rebooting of haywire components, or manual rebooting using commands that could potentially be relayed to the flight computer via an RF data link between the flight computer and the ground station.

This project represents a great leap forward in the development of IrishSat’s CubeSat prototype. Consolidating all of the necessary components for battery charging/discharging, voltage boosting and regulating, component switching, component biasing, and component monitoring on a single PCB will allow our prototype to become a more modular, self-contained system– relying on its own power and internal connections for operation, rather than external power and external connections. The software architecture that we developed similarly allows for this system to operate continuously and independently, allowing our prototype to become a complete embedded system unto itself and taking us one step closer to having a CubeSat ready for deployment.

## 2 Detailed System Requirements

The system will provide power to a multitude of sensors and components. The system needs to be able to integrate at least two solar cells with up to three LiPo batteries, allowing the batteries to be charged when there is sunlight available. Since the battery will output 3.7-4.2V, converters must be used to provide stable output voltages for different components. These include 1.8V for the IMUs, 3.3V for the ESP32, 5V for the Pi Zero, and 9V for the SDR and motor board.

The CubeSat needs to have a way to detect different states, including tumbling, light on solar panels, low battery level, and available window for downlink, which must be handled by our system. The tumbling state must be detected using an IMU that ProtoSat has selected, and the ESP32 must be able to recognize that tumbling state and send that information to the flight computer. The ESP32 also must use a component, like a photoresistor, to determine the light that the system is exposed to, in order to determine what types of components can be run efficiently. In order to fulfill this requirement, the ESP32 must also be able to turn off power-intensive devices, like the Raspberry Pi flight computer (Pi) and onboard software-defined-radio (SDR), when not in use. The energy of the battery must also be monitored by the ESP32 to determine when systems are safe to run, as discharging a LiPo battery too low can be extremely detrimental, so a component like a fuel gauge must be used to monitor this. The ESP32 must be able to communicate substantial information back and forth with the IMU and the fuel gauge. It must also be able to read analog data from the circuit used with the photoresistor; that photoresistor should be selected and that circuit should be designed so that the voltages for the light and no-light states are drastically different. Two IMUs should be implemented for redundancy, and the method of communication between them and the ESP32 and flight computer should take into account each device's different operating voltages.

The system will monitor the sensors to detect anomalies. In order to ensure proper operation of the circuit, the current flowing into the Pi and IMUs must be monitored, as well as the voltage provided by the 9V converter when it is active. Different specifications will be required for each device based on their expected data ranges. These readings must be stepped down as needed to provide no voltage higher than 3.3V to the pins of the ESP32. The ESP32's software must regularly check these readings and be able to recognize anomalies–that is, spikes in current or voltage–and send this information to the flight computer for eventual downlink. The ESP32 should also be able to respond to continuous anomalies by rebooting the device, or shutting it off entirely to conserve power. This software for this system constitutes the anomaly detection system.

On the hardware side, for this to function properly, enough pins on the ESP32 must be able to read analog voltages. To ensure protection of the ADC pins and a stable, reliable voltage sensing, a voltage-follower op amp should be used. This would protect the pins from voltage- and current-fluctuations. The analog voltage reading must be able to be converted by the software into a reading related to the actual measurement–i.e. the current into or voltage of the input power of a device. The shut-off circuit must be designed so as to not leave pins floating on the device, nor leave the device unconnected to ground. Such circuits should also not draw more power than the device does; these requirements may be met with a MOSFET configuration.

Since the ESP32 will be receiving all sensor data, it will be used to format and send data packets to the SDR. Our team will determine the data that is necessary to transmit and the frequency at which to sample the data. The ESP32 should not send all data to the flight computer so as to relieve the processing load required. The ESP32 should also be able to recognize when the flight computer and/or SDR are not powered on, and so it should not send any data. The ESP32 must be able to be reprogrammed by a user. The pins used for this are 3.3V, GND, EN, GPIO0, TX, and RX. In order that the device may properly select between download boot for programming and flash boot to run software from memory, GPIO0, TX, and RX should remain unused by the rest of the system.

The size limitations are substantial for this project. There is a separate PCB being designed by the IrishSat ProtoSat team that will control three motors used for attitude control. Our PCB will power this board, so we must send it power signals through a pin stack. Both PCBs and the SDR must stack to fit within a 5.5cmx6.5cmx4.25cm space, so the board should be no larger than 5.32cmx6.32cm. To satisfy this requirement, we will design our PCB with four layers, so we can utilize three separate layers to route traces. The PCB must not generate excessive noise that would compromise proper communication between devices. The PCB must have a ground layer, on which no traces may be routed, to ensure proper power and signal grounding. Components must only be placed on the top layer. Because many of the possible board manufacturers are incapable of making blind or buried vias, all vias must be through and intersect all layers of the board. In general, signal traces and pads should be at least 5 mil apart in order to avoid shorting two separate elements in the board manufacturing process. Trace width should be taken into account, and the power traces on the board from the batteries must be able to handle up to 1.4A. It must contain methods of contact for external components, including two-to-three batteries, two solar cells, a 9V board which will upconvert the battery voltage to 9V, a barrel plug for the SDR, the photoresistor (which must be sticking outside the CubeSat), the programming pins of the ESP32, and the Pi Zero flight computer. The external connectors must be arranged so that there is no overlap or collision when external devices are mounted on the board–this may compromise the system by not allowing all devices to connect with the PCB. The batteries, solar cells, SDR, photoresistor, and Pi Zero must be attached in such a way that both good electrical contact and secure mechanical connections are made. This precludes certain connection methods, such as Dupont wires, and guarantees some soldering must be done. PCBs are lightweight, so we are not concerned about exceeding any weight limits.

## 3 Detailed project description

### *3.1 System theory of operation*

Power is provided to the entire CubeSat through solar cells and LiPo batteries, and this board uses three batteries and two solar cells. During periods of sunlight, the solar cells charge the LiPo batteries, a process which is handled by the MPPTs (labeled “charger” on the block diagram). The remaining amount of battery energy left is also measured by fuel gauges, which relay that information to the ESP32 over I2C.

The LiPo batteries provide a variable voltage from 3.7-4.2V. Two batteries in series (providing 7.4-8.4V) are connected to a port which provides input power for the 1.8, 3.3, and 5V converters. The remaining battery is connected to a port which provides input power for the 9V converter. The power is distributed as subsystems require; the ESP32, fuel gauges, two current sensors, and one of the voltage-follower op-amps receive 3.3V, the two IMUs receive 1.8V, the Pi Zero, one current sensor, and four of the voltage-follower op-amps receive 5V, and the SDR receives 9V. The power lines of the Pi, IMUs, and the SDR are connected to MOSFET networks, so that the ESP32 can enable or disable those devices. The photoresistor is powered with a GPIO output on the ESP32. The ESP32 communicates with one IMU and the fuel gauges through an I2C bus which it is the master of. It also communicates with the Pi Zero and another IMU through a different I2C bus, which it is a slave of. To communicate properly, two level shifters convert the 3.3V I2C signals from the Pi and ESP32 to 1.8V signals for the IMUs. Current sensor readings for the IMUs and the Pi are received as analog signals. Those readings are each passed through a voltage-follower op-amp to ensure signal integrity. The 9V voltage is also monitored using a voltage-follower op-amp into an analog pin. Both the 9V analog reading and the Pi’s current sensor reading (which is referenced to 5V) are stepped down through voltage-divider networks before the op-amp.

When solar cells and batteries are connected, the ESP32 turns on and boots in flash mode. It turns on the 1.8V converter, the IMUs, and the fuel gauge, and waits to enter a tumbling state. This state would indicate that the satellite has been evacuated from the rocket and should begin running its main functions. The ESP32 then turns on the Pi Zero, joins its I2C bus, and provides power to the photoresistor.

After tumbling, the ESP32 switches between either a “day cycle” or “night cycle” depending on the reading from the photoresistor. Day cycle indicates there is enough light to charge the batteries through the solar cells, so the Pi remains on. Night cycle indicates there is not enough light, and to conserve power both the Pi (and the SDR, if it is on) are turned off. The SDR can be turned on by the ESP32 when the CubeSat is ready to communicate its data, but to conserve power, it is off by default.

Meanwhile, the ESP32 monitors the currents of the IMUs and the Pi and monitors the 9V line. Any “spikes” in current or voltage are noted and relayed to the Pi. Spikes may indicate malfunction, so these devices may be restarted or turned off indefinitely.

## 3.2 System Block diagram

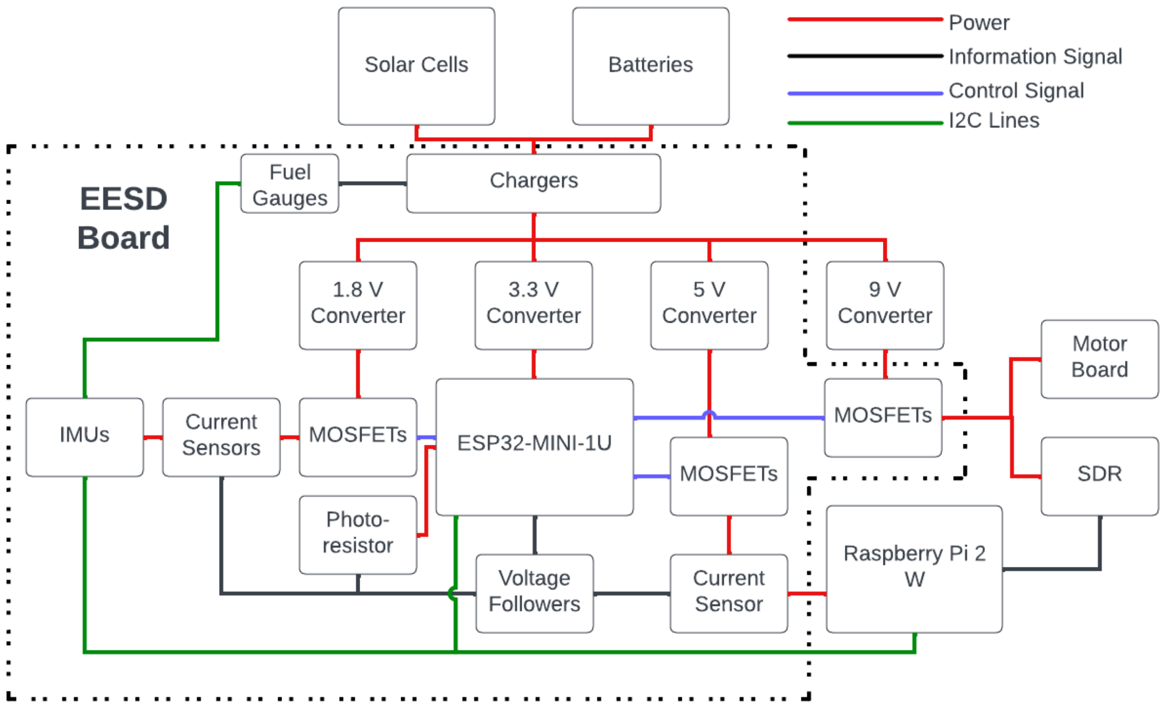


**Figure 6.** System Block Diagram

## 3.3 Power

The power subsystem is responsible for powering the 1U CubeSat prototype, including the Senior Design PCB, a Raspberry Pi Zero 2 W, a Motor Controller Board (designed by the ProtoSat team on IrishSat), and three reaction wheel motors. These components have a variety of power needs.

The power is generated in our subsystem using four solar cells. The most important consideration when choosing a solar cell was size. The solar cells are 11cm long and 6.6cm wide, which is within the 1U size range specified by NASA in the CSLI Guidelines. Only four of these solar cells can fit on the 1U CubeSat, so size also plays a role in the quantity of solar cells used. The maximum output of each solar cell is 6.07V and 200mA, resulting in max 1.22W. Two solar cells are wired in parallel, resulting in two sets of solar cells, each outputting max 2.44W. The power output changes with the intensity of sunlight exposure. The model number is P124 R1G and is produced by Voltaic.

Wiring our solar cells in parallel was a decision driven by size constraints. Our CubeSat cannot fit four batteries, but it can fit four solar cells. Having two sets of solar cells allowed for the utilization of only two battery chargers and two batteries. Wiring all four in parallel would be a critical source of failure and would only allow for one battery, which is not ideal for storing maximum power.

This subsystem called for two 3.7V LiPo batteries. In practice, three LiPo batteries were used due to unforeseen PCB design challenges, which will be discussed at the end of this section as well as in the To-Market Design Changes section. These LiPo batteries were each 6cm long and 3.6cm wide. The charge cutoff voltage is 4.2V. Their capacity is 2Ah each. Their max continuous discharge current is 1A each, which allows for a total max current of 1.4A when adding the current of one solar cell set. Our system uses two of these sets, creating a total max current of 2.8A for the system.

Our system is divided into two separate power sets. The power set is defined as one LiPo battery and two solar cells in parallel that are managed by the battery charger. Power Set 1 is defined as containing Battery 1 (BT1) and Solar Cell 1 (VSC1) on the PCB silkscreen. Power Set 1 powers the 1.8V, 3.3V, and 5V power rails. Power Set 2 is defined as containing Battery 2 (BT2) and Solar Cell 2 (VSC2) on the PCB silkscreen. Power Set 2 powers the 9V power rail. The reason for splitting up the power system is because placing the battery chargers in parallel caused a multitude of issues. First, the battery chargers can charge each other, which could damage the componentry in the charger and overcharge the batteries. To remedy this issue, Schottky diodes were placed in series with the load outputs of the chargers to prevent the reverse current.

The issue with this method is that the battery chargers did not evenly share the load. In testing, one battery charger would output a large amount of current while the other would output a small amount of current (~10mA). This outcome was predicted in online research, but battery charger breakout boards in series with Schottky diodes were used to test. In this layout, one power set would be neutralized, causing a large decrease in total system power. The solution was using the power sets for different power rails. The power rails were split up based on their current limits. The 9V rail supplies power to the reaction wheel motors and the Sidekiq Z2 SDR. The reaction wheel motors need to pull as much current as possible, so there should be no other essential components on that rail to drain current. The SDR can be turned off and on by the ESP32, so our system can determine if Power Set 2 has enough charge to turn on the SDR. The SDR is an essential component, but our system can always wait to transmit back to the Ground Station until there is enough charge to transmit and hold its orientation. The reaction wheels will also not be running continuously, since they should stabilize the system fairly quickly. Power Set 1 handles the other voltage rails (1.8V, 3.3V, and 5V) which power the ESP32, Raspberry Pi Zero 2 W, and other necessary componentry.

The battery charger is a 1-Cell 1.5A Linear Battery Charger (BQ24074). The battery charger features dynamic power path management (DPPM) that powers the system while simultaneously and independently charging the battery. The battery charger prioritizes powering the system over charging the battery, lowering the number of charge and discharge cycles on the battery. The charger allows the system to run with a defective or absent battery pack, supplying power solely from the solar cells. The battery charger reads the battery voltage and terminates battery charging once it reaches its maximum charge (4.2V). The max voltage load output of the battery charger is regulated to 4.4V. If the solar panels are not receiving any light, the charger will output the same voltage as the battery. If the battery is dead or not connected, the charger will output the same voltage as the solar cell. If the solar cells output over 4.4V, the battery charger load output will regulate it to 4.4V. The battery charger has a maximum load output current of 5A, which is over the maximum source current of 1.4A from one LiPo and two solar cells in parallel. The maximum charging current of the LiPo is 1.5A.

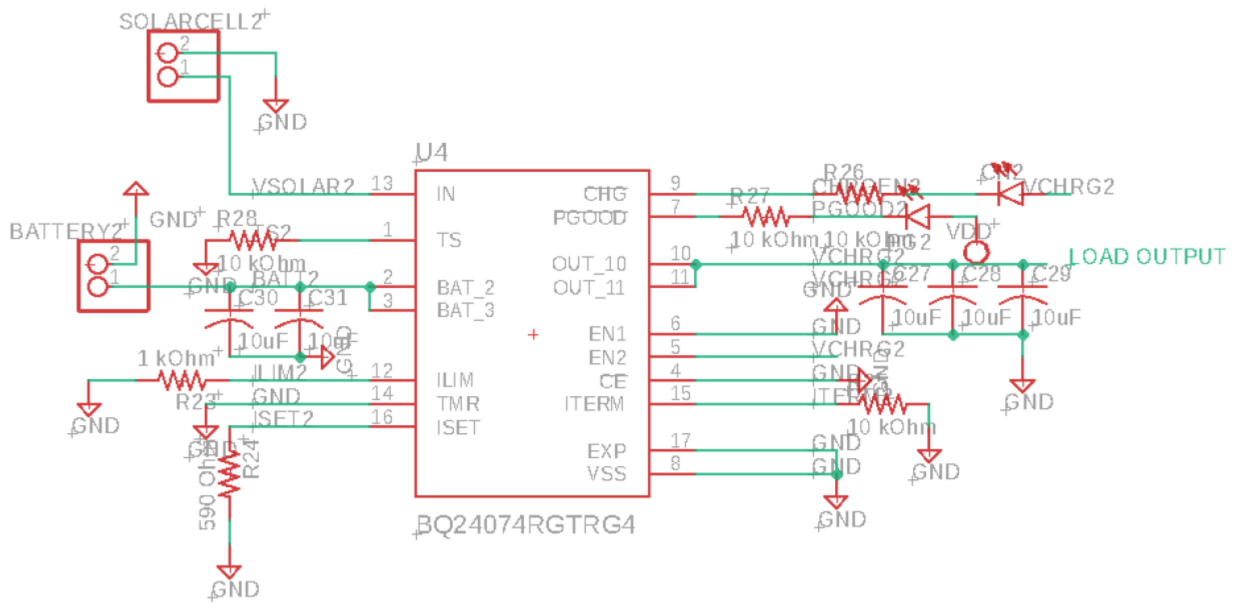


**Figure 7.** Battery Charger

Since the battery charger outputs a range of voltages with a maximum value of 4.4V, the load output needs to be regulated to power multiple different voltage rails. Our system utilizes four DC-DC converters instead of linear voltage regulators. For the 5V and 9V power rails DC-DC boost converters must be used. For the 1.8V and 3.3V power rails, linear voltage regulators were an option. Linear voltage regulators lack efficiency. The input current essentially equals the output current. Since voltage is decreasing and current is remaining equal, total power is lost. A DC-DC converter can increase current in proportion to decreasing voltage to keep power essentially equal. DC-DC converters usually have high efficiencies around 95%, meaning 5% of the input power will be lost to the converter. DC-DC converters do not have the same output voltage accuracy of linear voltage regulators, but for this application the improved accuracy is not necessary.

The two IMUs (ICM-20948) require 1.8V. The 1.8V DC-DC converter (XCL210) was used to supply this voltage. The max current consumption for the 1.8V rail is 6.22mA. The XCL210 has a max output current of 200mA with an efficiency of 93%. These specifications exceeded the power requirements for this rail. The XCL210 also has an Enable pin for easily switching the converter off or on. This rail was powered directly by Power Set 1 from the output of the battery charger.

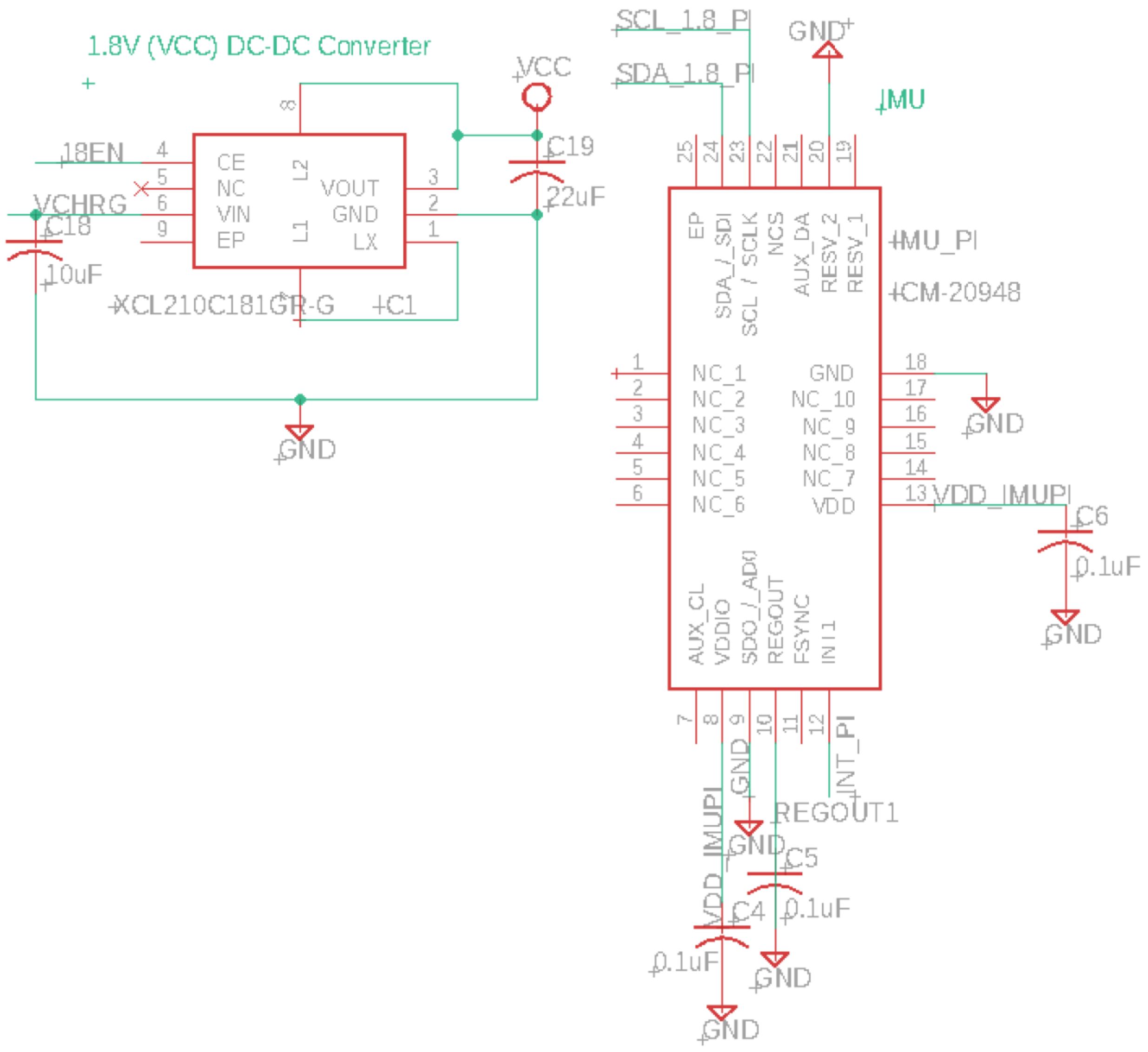


**Figure 8.** 1.8V DC-DC Converter and IMU

The ESP32-MINI-1U-H4, two battery fuel gauges (MAX17043), four LEDs (PG1, CH1, PG2, and CH2), two current sensors (ACS725LLCTR-10AB-T), and a low-power op amp (LM321MF_NOPB) require 3.3V. The 3.3V DC-DC converter (RPM3.3-2.0) was used to supply this voltage. The max current consumption for the 3.3V rail is 793.3mA. The RPM3.3-2.0 has a max output current of 2A with an efficiency of 98%. These specifications exceeded the power requirements for this rail.

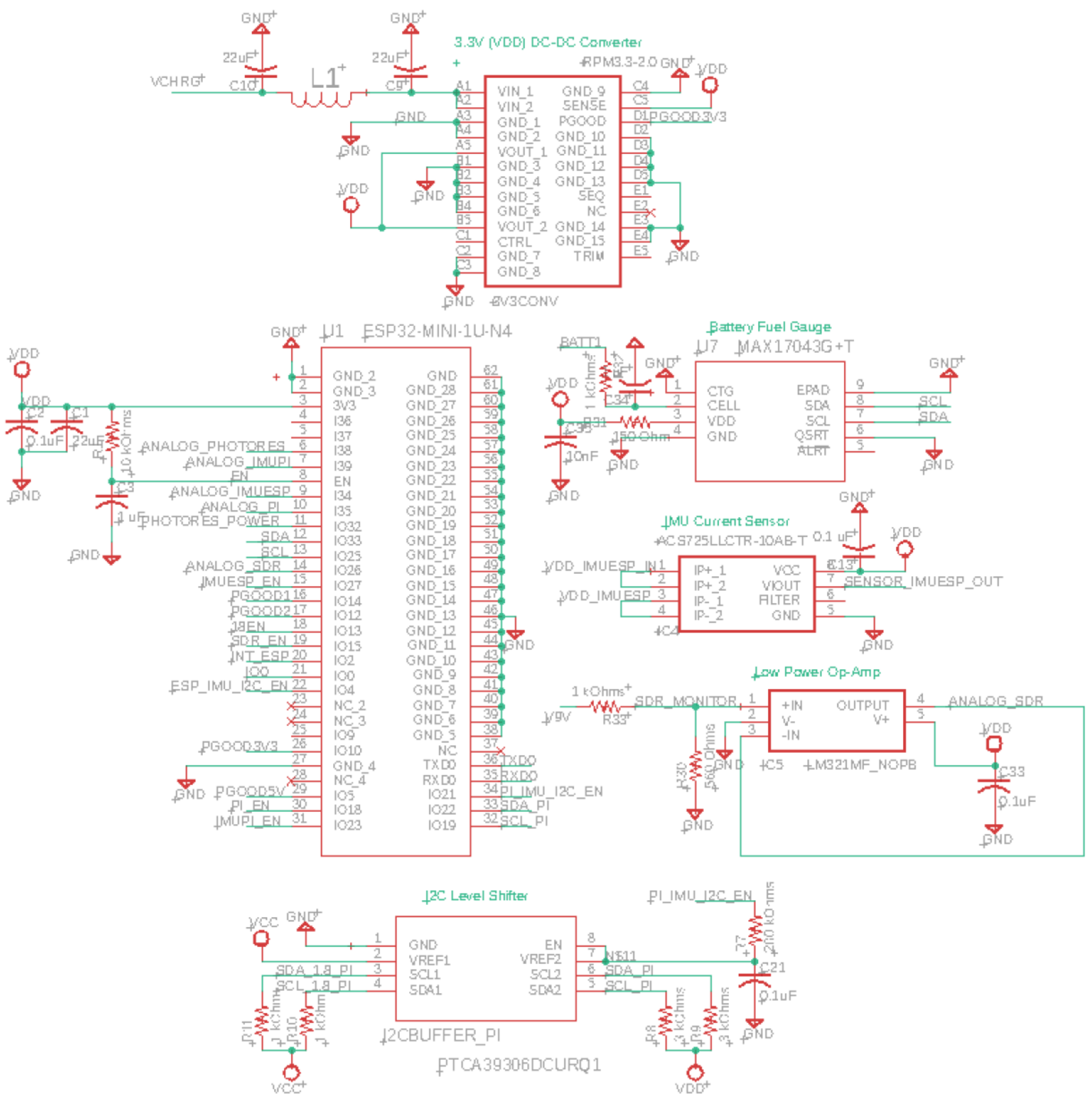


**Figure 9.** 3.3V DC-DC Converter, ESP32-MINI-1U-H4, Battery Fuel Gauge, IMU Current Sensor, I2C Level Shifter & Low Power Op Amp

The Raspberry Pi Zero 2 W, a current sensor (ACS724LLCTR-10AB-T), a unity gain op amp (LMV934MA), and an LED controller on the Motor Controller Board (PCA9685) all require 5V. The max current consumption for the 5V rail is 414.84mA. The 5V DC-DC converter (RPM5-2.0) was used to supply this voltage. The RPM5-2.0 has a max output current of 2A with an efficiency of 98%.

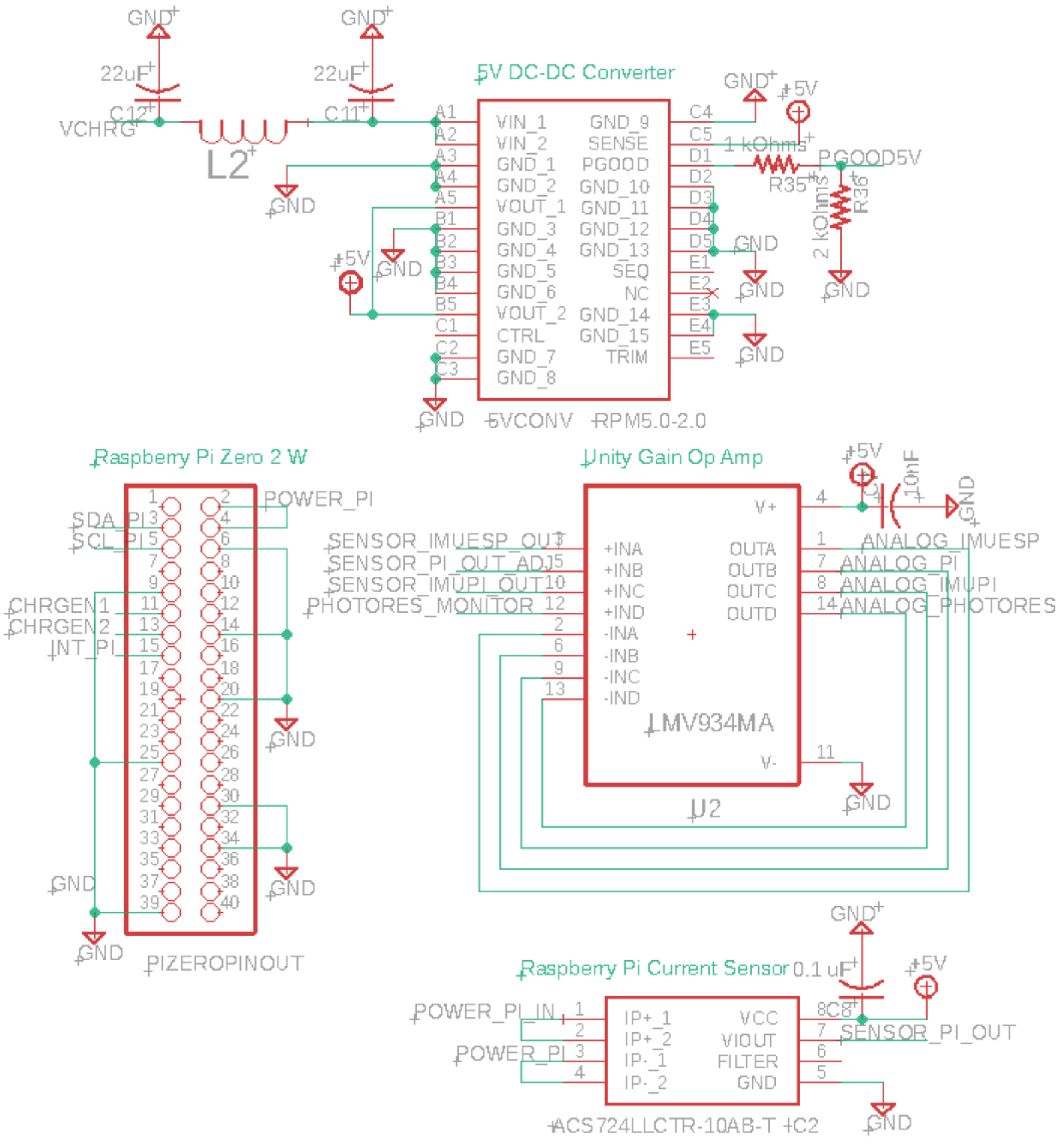


**Figure 10.** 5V DC-DC Converter, Raspberry Pi Zero 2 W, Raspberry Pi Current Sensor, Unity Gain Op Amp

Our system design has an error in utilizing this 5V DC-DC converter. This converter can only step down voltage, although the data sheet claims its input voltage range is 3-17V. To remedy this mistake, two 3.7V LiPos were connected in series to increase the output voltage to 7.4V. The RPM5-2.0 would then step down the voltage to 5V successfully. This design change does not allow for battery charging, since the

BQ24074 battery charger only allows for a single cell battery input. The batteries in series could also charge unevenly, creating a potential for overcharging one of the batteries. Due to these reasons, Power Set 1 cannot not charge the batteries in series. Power Set 2 maintained full functionality of charging the LiPo battery while also powering the 9V rail.

The reaction wheel motors and the Sidekiq Z2 SDR require 9V. The SDR requires around 330mA maximum when transmitting. As mentioned above, the SDR will be completely off when not in the Downlink state, meaning all 1.4A from Power Set 2 will be used to power the reaction wheel motors. When converting from 4.4V to 9V, the output current is roughly half of the input current, since the power remains equal, meaning the reaction wheel motors will realize around 700mA maximum. The reaction wheel motors at full speed draw around 400mA total. At max acceleration, the current draw can jump to 850mA. The Motor Controller Board will have to limit the acceleration of the reaction wheels to keep the current draw below 700mA. Limiting acceleration will limit the functionality of the attitude control system, but it will still be effective.

The 9V DC-DC converter (U3V40F9) was used to supply this voltage. This converter is its own PCB and was connected to the Senior Design PCB using a pinstack. The reason for not transferring the converter on to our PCB was because of its small size and the lack of space on our PCB. The 9V converter board is only 1.5x1.5cm. Since the Raspberry Pi Zero 2 W also connects to our PCB through a pinstack, this converter sits right next to it without taking up any more height. Otherwise, there would have been an open gap next to the Raspberry Pi Zero 2 W that the 9V converter now fills. The 9V converter handles continuous input currents up to 3.5A, meaning it easily handles the max output current of 1.4A from Power Set 2.

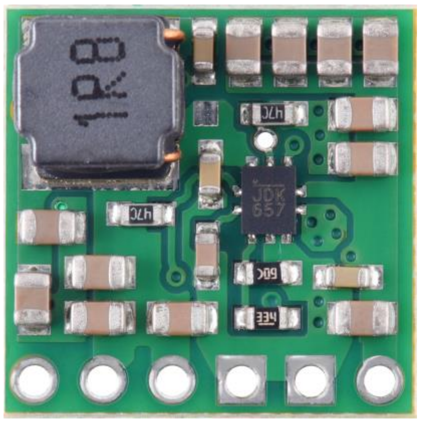


**Figure 11.** 9V DC-DC Converter

**Table 1.** Power Draw by Component

| Component | Bias Voltage (V) | Typical Current Draw (mA) | Max Current Draw (mA) | Typical Power Draw (mW) | Max Power Draw (mW) |
|---|---|---|---|---|---|
| **Power Set 1** | | | | | |
| ICM-20948 IMU (x2) | 1.8, 1.8 (VDDIO) | 3.11 | 3.11 | 5.598 | 5.598 |
| | **1.8V Rail Total** | 6.22 | 6.22 | 11.196 | 11.196 |
| ESP-32 | 3.3 | 20 | 108 | 66 | 356.4 |
| Current Sensors (x2) - ACS725LLCTR-10AB-T | 3.3 | 10 | 14 | 33 | 46.2 |
| Level Shifters (x2) | 3.3 | 64 | 128 | 211.2 | 422.4 |
| Battery Fuel Gauges (x2) - MAX17043 | 3.3 | 0.05 | 0.075 | 0.165 | 0.2475 |
| LEDs (x4) | 3.3 | 30 | 100 | 99 | 330 |
| Single Op Amp - LM321MF/NOPB | 3.3 | 0.430 | 1.15 | 1.419 | 3.795 |
| | **3.3V Rail Total** | 288.53 | 793.3 | 952.149 | 2617.89 |
| Raspberry Pi Zero | 5 | 120 | 400 | 600 | 2000 |
| Quad Input Op Amp - LMV934MA/NOPB | 5 | 0.464 | 0.840 | 2.32 | 4.2 |
| Current Sensor - ACS724LLCTR-10AB-T | 5 | 10 | 14 | 50 | 70 |
| | **5V Rail Total** | 130.464 | 414.84 | 652.32 | 2074.2 |
| | **Power Set 1 Total** | 425.214 | 1214.36 | 1615.665 | 4703.286 |

| Power Set 2 | | | | | |
|---|---|---|---|---|---|
| Sidekiq Z2 (SDR) | 9 | 228 | 620 | 2052 | 5580 |
| Motors | 9 | 400 | 700 | 3600 | 6300 |
| | **9V Rail Total** | 628 | 1320 | 5652 | 11880 |
| | **Power Set 2 Total** | 628 | 1320 | 5652 | 11880 |
| | **System Total:** | 1053.24 | 2534.36 | 7267.665 | 16583.286 |

## *3.4 Anomaly and State Detection*

The Anomaly and State Detection subsystem is required to detect and respond to the current state of the CubeSat as it is in orbit, as well as to monitor the current draw and/or voltage of all high power components, ensuring that devices are operating correctly and within recommended tolerances. If devices are not needed in the system's current state, or if they are drawing too much power/voltage, a MOSFET network must be able to switch the device on or off using a 3.3V enable signal from the ESP32, regardless of device bias voltage. The overall operation of the subsystem should be to ensure that power is consumed by the full system in the most efficient way possible, depending on both internal and external conditions of the CubeSat.

The current sensors chosen for our board were the ACS724LLCTR-10AB-T and the ACS725LLCTR-10AB-T, both from Allegro Microsystems. These were chosen because they fulfill the voltage and current requirements for their respective components. The ACS724LLCTR-10AB-T is designed for 5V, and the ACS724LLCTR-10AB-T is designed for 3.3V, which allowed these to measure current from the Pi and the IMU's, respectively. Both can handle 10 A of current, which is well beyond the current inputs to either the Pi or the IMU's. Both current sensors output an analog voltage signal which exhibits a linear response with respect to input current, allowing the ESP32 to map the voltage reading to current using a simple linear formula. An example from the ACS724 is shown in **Figure 12**, reproduced from Allegro's datasheet**.** The sensitivity of the ACS725 is 132 mV/A, of the ACS724 200 mV/A. The zero value is 1.65V; functionally no device should drop below this, as it would indicate current in the reverse direction.

The output of each current sensor is passed through a voltage-follower op-amp. These op-amps protect the ADC from voltage- or current-fluctuations by providing a low output impedance and a stable, reliable voltage.

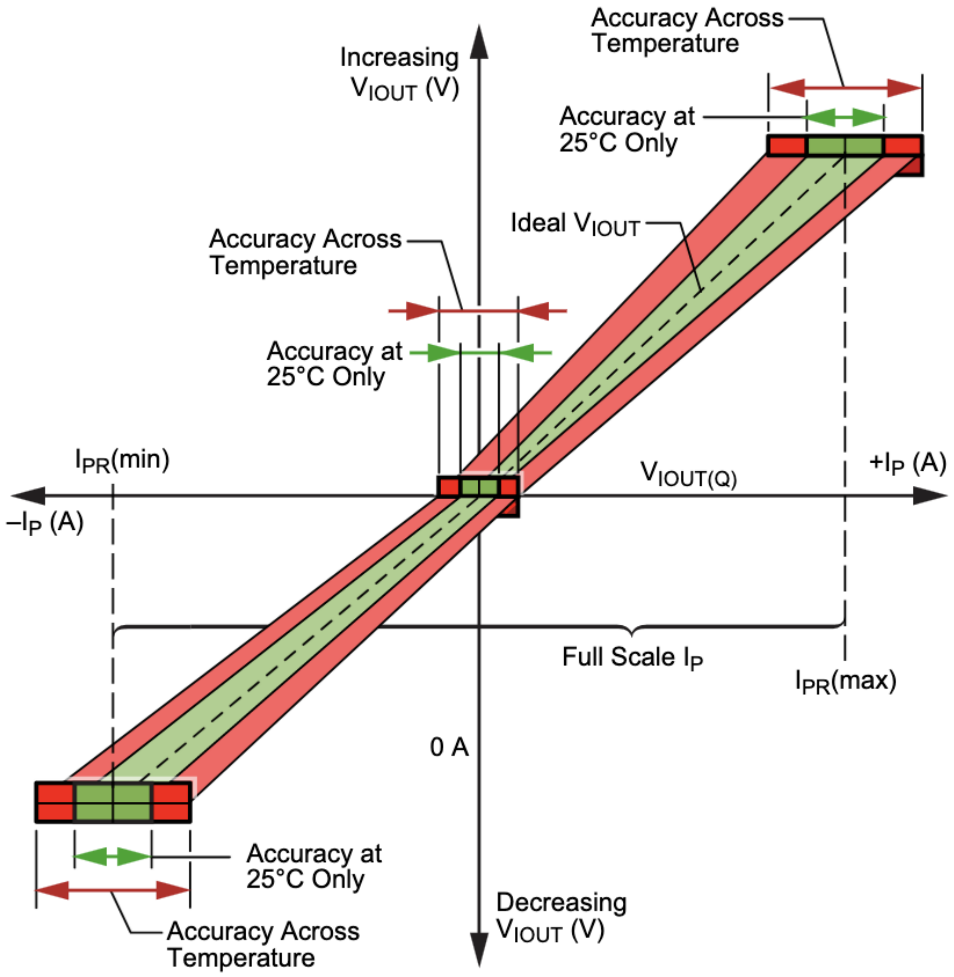


**Figure 12.** The response of the ACS724, reproduced from Allegro's datasheet. The output voltage is linearly proportional to the current, and changes with temperature.

The MOSFETs chosen to effect the switching circuit were the FDN86501LZ and the FDN352AP, which are an N-Channel FET and a P-Channel FET, respectively. The FDN86501LZ was chosen because it has a low threshold voltage ($V_t$ = 2.4V max, 1.9V nominal). This ensures that it can be turned on by a 3.3V enable signal from the ESP32, as the equation for NMOS saturation is $V_{gs} > V_t$ , and the source voltage for all N-Channel MOSFETs in this configuration is 0V (ground). The FDN86501LZ is also able to handle 60V and 2.6A, which for all configurations is sufficient to handle required voltage and current draw. The FDN352AP was chosen for its relatively low threshold voltage ($V_t$ = -2.5V max, -2.0V nominal), high voltage tolerance (-30 $V_{ds}$ ), and high drain current tolerance (-1.3 A).

It was originally conceived of to design four separate MOSFET networks, with the additional two being used to switch on and off the Pi's IMU and the ESP32's IMU separately, but as it was realized that it would not be necessary to have one IMU on without the other, and additionally that the IMU's draw negligible current in practice, these other two MOSFET networks were removed from the final design. The ability to switch on and off IMU's was retained through the ESP32's direct control over the 1.8V DC-DC converter, which requires a high enable signal to operate. As it is not necessary to have control over the power to individual IMU's, writing high or low to the 1.8V DC-DC converter enable pin allows control over the entire 1.8V net, which allows the system to turn on and off both IMU's together.

The two MOSFET networks that were retained control power output to the Pi flight computer and the SDR. To implement these systems, which use a 3.3V enable signal from the ESP32 to switch 5V and 9V signals to the Pi flight computer and SDR, respectively, the following MOSFET topology was used (shown in **Figure 13.**).

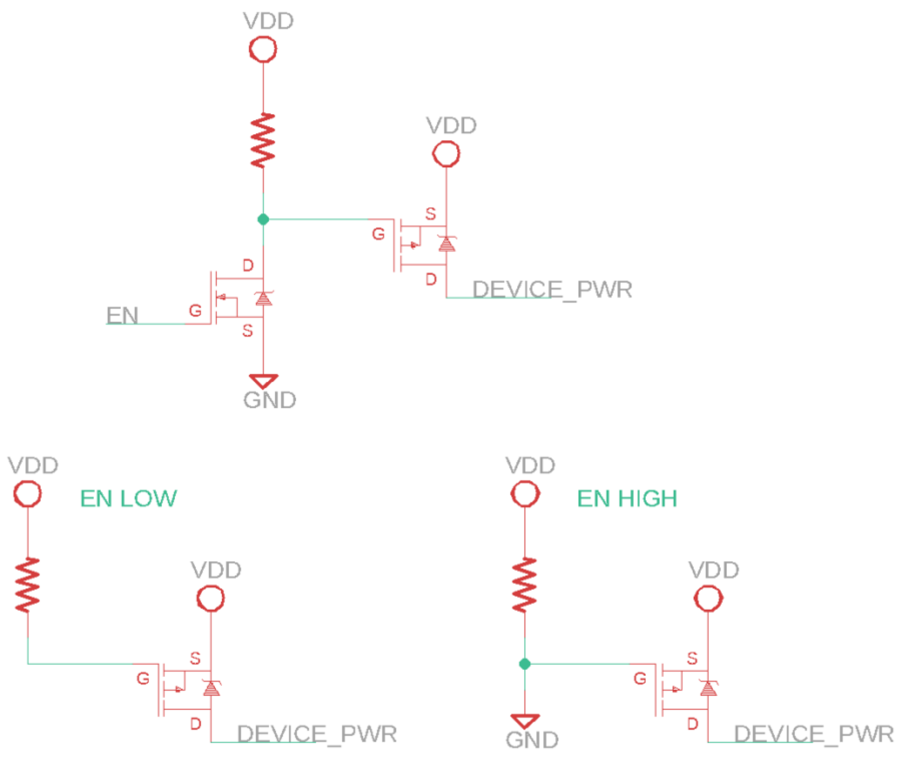
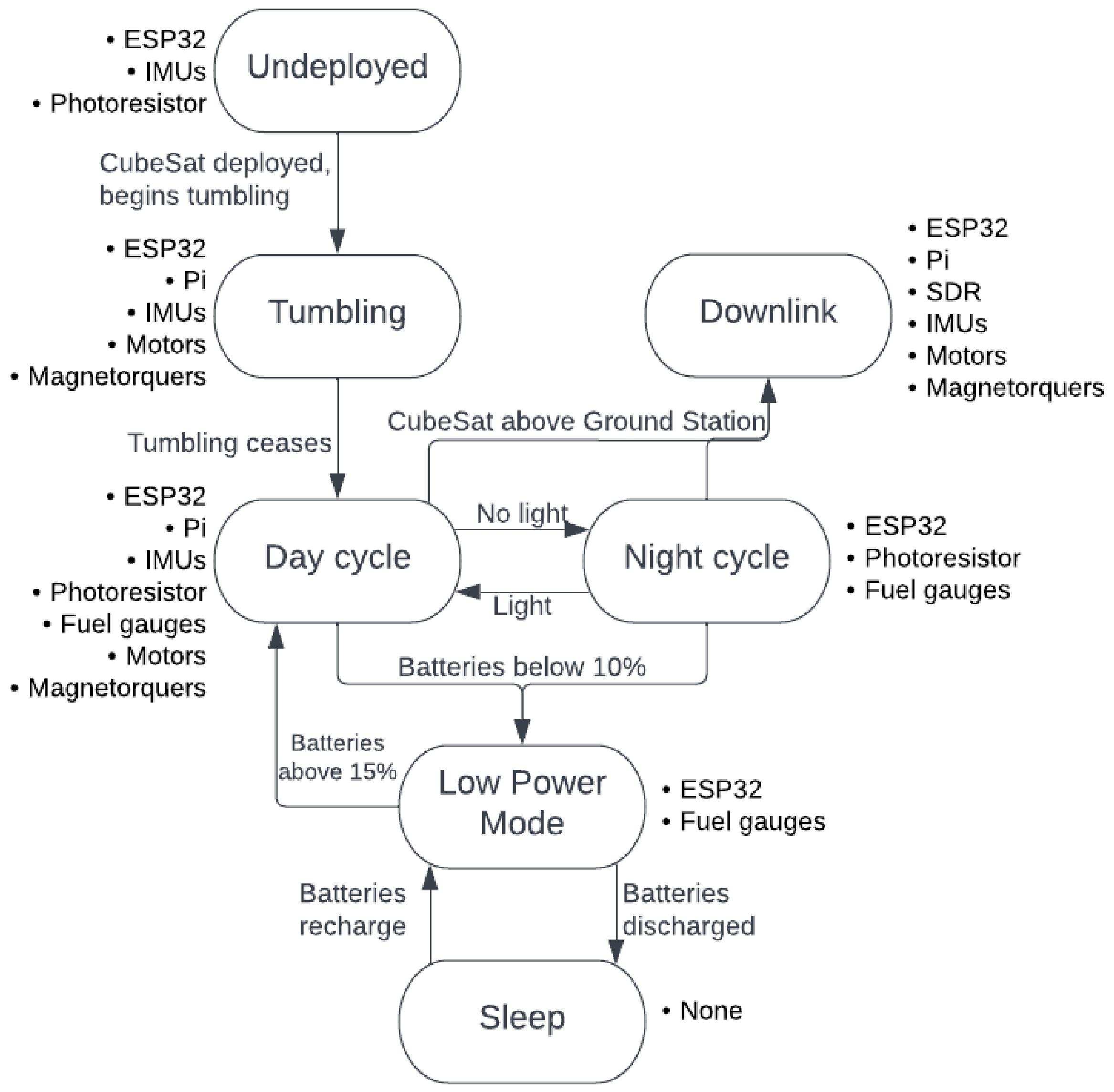


**Figure 13.** The MOSFET circuit topology used. The "effective" circuits for when the enable signal is low and high are also shown.

This circuit works by routing an enable signal from the ESP32 to the gate of an N-Channel MOSFET. This N-Channel MOSFET has its source connected to ground and its drain connected to the gate of a P-Channel MOSFET in parallel with the low side of a 10 kΩ resistor that is in turn connected to VDD. The source of this P-Channel MOSFET is connected to VDD, while the drain is connected to the input pin of the device under test. When the 3.3V enable signal from the ESP32 is low, the N-Channel MOSFET is an open circuit, meaning that a full 9V is present at the low end of the 10 kΩ resistor, as negligible current flows into the gate of the P-Channel MOSFET. Since for a P-Channel MOSFET, cutoff condition is reached when $V_{sg} \leq |V_t|$, VDD present at the gate will result in a low voltage being present on the drain, and the device being turned OFF. When the 3.3V enable signal from the ESP32 is high, the N-Channel MOSFET is an closed circuit, resulting in ground being present at the gate of the P-Channel MOSFET. $V_{sg}$ of the P-Channel MOSFET will be equal to 0, and since the saturation condition for a P-Channel MOSFET is $V_{sg} \geq |V_t|,$ this will result in the P-Channel MOSFET presenting VDD to the input pin of the device, and the device will be turned ON.

The following section discusses the detailed operation of the subsystem's different states. A finite state machine diagram is shown in **Figure 14**, showing the transition conditions between states and the enabled devices in each state (assuming full CubeSat functioning). This finite state machine was implemented in code using the functions checkState() and updateState(), both contained within Tumbling.cpp (See Appendix B, 9.2 ii.).



○

**Figure 14.** Finite state machine representation of the system.

*Undeployed*

The Undeployed state describes the state of the CubeSat prior to ejection from the rocket. This is a meta-state that will prohibit entry into any other state besides "Tumbling". This state is determined by the fact that tumbling has not occurred yet. In this state, everything is turned off except for the ESP32 and the IMU's.

*Tumbling*

The tumbling state represents the state of the CubeSat immediately after being injected into an orbit, when it will be “tumbling” through space with constant angular velocity. This angular velocity is detected by the ESP32’s IMU, and the algorithm to detect CubeSat tumbling is implemented on the ESP32, reading IMU data over an I2C connection. The IMU chosen was the ICM-20948. This IMU was chosen because it is the world’s lowest power 9-axis IMU, which combines a 3-axis gyroscope, 3-axis accelerometer, 3-axis compass in a 3x3x1mm package. This allows the system to measure the absolute orientation of the CubeSat as well as the angular velocity during tumbling.

Algorithmically, tumbling detection is done in the following manner: I2C signals from the ESP32’s IMU are read by the ESP32. The angular acceleration from the gyroscope is read and stored in a variable. If this angular acceleration in either the x, y, or z axis exceeds a threshold of 1 radians/sec for longer than 3 seconds continuously, the system is considered to be “tumbling”, and the state variable is updated accordingly. This detection is implemented in code as the tumblingDetection() function, contained in Tumbling.cpp (See Appendix B, 9.2 ii.).

In this state, the ESP32 turns on the Pi flight computer. After full integration with the Proto-Sat team’s motor board, the attitude control system (controlled by the flight computer and contained externally on the motor controller board) will attempt to slow the rotation of the CubeSat.

*Day Cycle*

The Day Cycle represents regular operation of the CubeSat during the “day”, or the period of time in which it is exposed to the sun, has sufficient power, and is not downlinking. Detection of a voltage above a certain threshold across a photoresistor arranged in a voltage divider circuit will be used to enable this state. The photoresistor chosen was the PDV-P8104. This photoresistor was chosen because of a favorable difference between its “OFF” resistance and its “ON” resistance, which enabled greater voltage resolution in reading the voltage divider circuit using an analog input from the ESP32. Below is a schematic representation of the circuit.

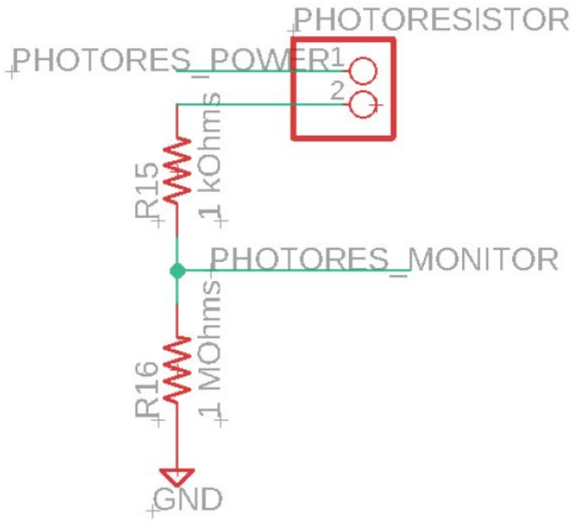


**Figure 15.** The photoresistor monitoring circuit. PHOTORES_POWER is the output of the ESP32's GPIO32; the signal PHOTORES_MONITOR passes through a voltage-follower op-amp before being monitored by the ESP32's I38, which functions as an ADC pin.

The circuit was designed in order to yield two largely different voltage readings. Specifically, the voltage values desired were about 1.0V when dark and just under 3.3V when light. While illuminated, the photoresistor has a resistance of 27-60 kΩ, resulting in a voltage of about 3.1-3.2V on the monitor pin. While dark, the photoresistor has a resistance of 2 MΩ, resulting in about 1.1V on the pin. This is a difference of about 2V; since the ESP32 12-bit ADC has a resolution of about 1240/V, the two states are separated by about 2480 analog values. This circuit is shown in **Figure 15.**

Day Cycle determination in code is done by the function dayCycleCheck(), contained within Tumbling.cpp (See Appendix B, 9.2 ii.). In this state, everything is enabled but the SDR. The SDR is not necessary here because there is no transmission.

*Night Cycle*

The Night Cycle represents regular operation of the CubeSat during the "night", or the period of time in which it is not exposed to the sun, has sufficient power, and is not downlinking. Reading of a photoresistor monitor voltage below the "day" threshold enables this state. In this state, the, SDR, IMU, and Pi flight computer, leaving only the ESP32 fully operational. Since there is no sun to charge the solar panels, the attitude control system is unnecessary, meaning that the Pi and IMU can be shut off. If there is an available window for downlink, the downlink state can be entered provided that the battery has sufficient power. Night Cycle determination in code is made by a false output from the function dayCycleCheck(), contained within Tumbling.cpp (See Appendix B, 9.2 ii.).

*Downlink*

In the downlink state, the SDR will be turned on using an enable signal to the SDR MOSFET network. This state will be entered into based on the readings of a GPS module on the completed CubeSat. The current iteration of the system does not integrate a condition for entering downlink state based on an external signal, but since we have demonstrated the ability to power the SDR, to switch it on and off using I2C signals from the flight computer, and also to form data packets containing information about anomalies and battery level, it is reasonable to state that implementing a downlink state using our current system setup will be easily accomplished with modifications to software only.

*Low Power*

The Low Power state is entered when the charge level of the battery falls below a certain threshold (10%). This condition will be detected by the battery fuel gauge and the state triggered by the ESP32. In the low power mode, everything will be shut off except for the battery fuel gauge and the ESP32, which will also place itself into sleep mode. This state is exited upon batteries reaching 15% charge. Since our board had several unforeseen issues with our selected fuel gauges (SDA and SCL lines were flipped, both gauges had the same I2C slave address, documentation was poor), we did not successfully implement low power detection in code. However, it is reasonable to think that if we selected different fuel gauges, it would have been possible to implement a low power check function within our finite state machine similarly to how we implemented dayCycleCheck().

*Sleep*

The sleep state represents a last-ditch attempt to save the batteries of the CubeSat if their charge level falls below a dangerous threshold. Fully discharged LIPO batteries have been known to become permanently damaged, and thus it is in the best interest of the power system to avoid over-discharging them. Since LIPO batteries will output less than nominal voltage if they are below a certain charge threshold, upon reaching a dangerous level of discharge the LIPO will naturally output less than 3.3V, resulting in a bias voltage that is too low to run the ESP32. Since we use a combination of NMOS and PMOS in our MOSFET networks, a floating voltage on the NMOS gate after ESP32 shutdown will result in a low output of the PMOS. This ensures that upon

ESP32 failure, our MOSFETs do not simply switch on all components and thereby drain what little power remains in the battery.

The following section discusses anomaly detection, which was accomplished by measuring input current to the Pi flight computer, the ESP's IMU, and the Pi's IMU, as well as input voltage to the SDR. Current readings from the current sensors are made indirectly by first measuring an analog volage output from the sensor and then mapping to current within the code. Voltage readings were made by measuring voltage through a voltage divider (to avoid overloading the analog input pins) and then mapping to true voltage in code. Anomaly detection for the SDR was done using the checkAnomalies function contained in Tumbling.cpp, while anomaly detectction for the Pi and the IMU's was done using the measure_imu_currents() function contained in sensors.cpp (See Appendix B, 9.2 ii.).

Data processing for each analog input signal was tailored for each device in order to detect anomalies for the specific device. The ACS725s, connected to the IMUs, are expected to reach only about 1.67V or so under normal operation. Before the voltage-follower, the output reading of the ACS724 goes through a voltage divider using 1 kΩ and 2 kΩ resistors, which attenuates the signal by a factor of ⅔. This reduces the max voltage from 5V to about 3.4V, which is acceptable for the ESP32's pins. The Pi Zero's input current is stable from around 100-200mA, which is indicated by about 1.7-1.8V on the ADC pin. If the Pi is on and the voltage drops below or rises above these values, the ESP32 reports a negative or positive current spike, respectively.

The voltage from the 9V converter is also monitored. The voltage is first passed through a voltage divider using 1 kΩ and 560 Ω resistors, which attenuates the signal by a factor of about 0.36. This means the max voltage into the op-amp is about 3.23V, which is acceptable for the ESP32's pins. This voltage, like the current sensor readings, is passed through a voltage-follower op-amp to ensure signal integrity before reaching the ESP32 pin.

## *3.5 Flight Computer Communication*

The major system requirement for the communication with the ProtoSat flight computer is that because the Pi Zero is the flight computer and the ESP controls a subsystem of the CubeSat, the Pi must be in control of when it receives the ESP reports. Therefore, the ESP had two I2C buses; one that the ESP controlled to communicate with the IMU and fuel gauges that it controlled, and one that connected the ESP to the Pi that the Pi controlled.

The ESP32 joins the TwoWire PiBus with an address of 0x08, and registers the requestEvent() and receiveEvent() functions in the PiComm library to respond to request and receive events on that bus. The receiveEvent() function is triggered when the Pi writes to the ESP32, and the requestEvent() function is triggered when the Pi reads from the ESP32. If the Pi is requesting data from the ESP32, it must first write to the ESP32 to inform it of its request, so that when the requestEvent() function is triggered, the ESP32 knows what data to return to the Pi.

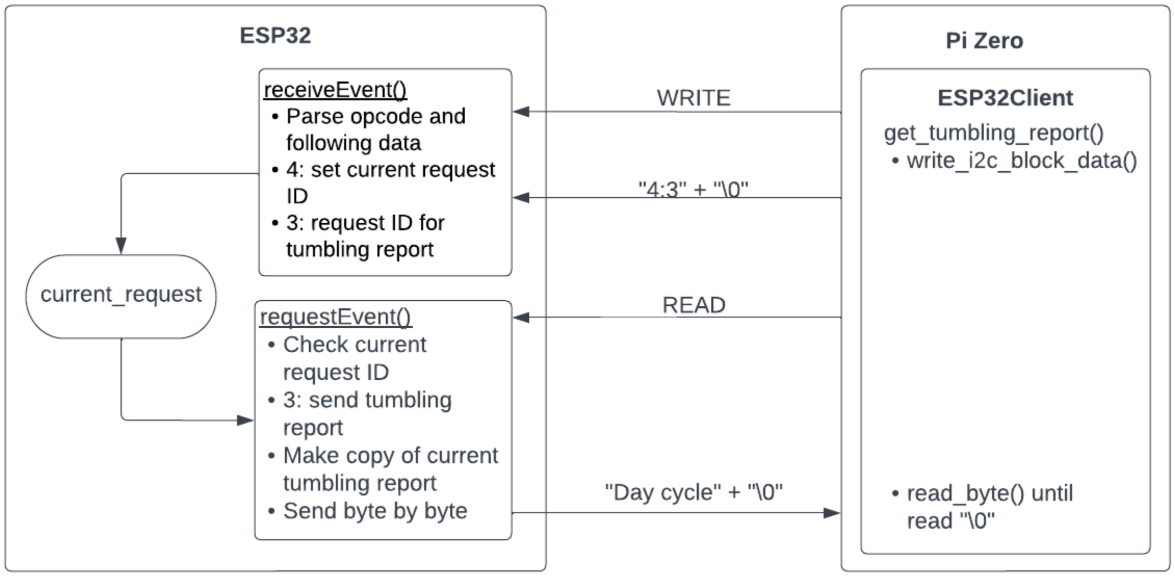

ESP32
receiveEvent()
• Parse opcode and following data
• 4: set current request ID
• 3: request ID for tumbling report
current_request
requestEvent()
• Check current request ID
• 3: send tumbling report
• Make copy of current tumbling report
• Send byte by byte
WRITE
"4:3" + "\0"
READ
"Day cycle" + "\0"
Pi Zero
ESP32Client
get_tumbling_report()
• write_i2c_block_data()
• read_byte() until read "\0"

**Figure 16.** Interaction between the ESP32 and the Pi Zero

If the Pi is sending a command to the ESP32, it can simply write to the ESP32. Because all writes and reads begin with a write that triggers the receiveEvent() function, the nature of the writing is determined by the opcode sent. Messages are sent in the format of “opcode:data”, where the opcode is the function requested and the data is the necessary data to complete that function. For example, the opcode to set the current request ID is 4, so a write of “4:3” will cause the receiveEvent() function to set the current request ID to 3, which is the request ID to send a tumbling report. Then, when reading from the ESP32, the requestEvent() function has a switch statement that will call the send_tumbling_report() function if the current_request is 3. The opcodes were chosen to reduce communications over the I2C, and are abstracted in the Pi with the ESP32Client class, with the following methods:

- get_anomaly_report()
- get_tumbling_report()
- restart_sensor()

- shutdown_sensor()
- get_current_sensor_readings()

See Appendix B for the complete ESP32Client class.

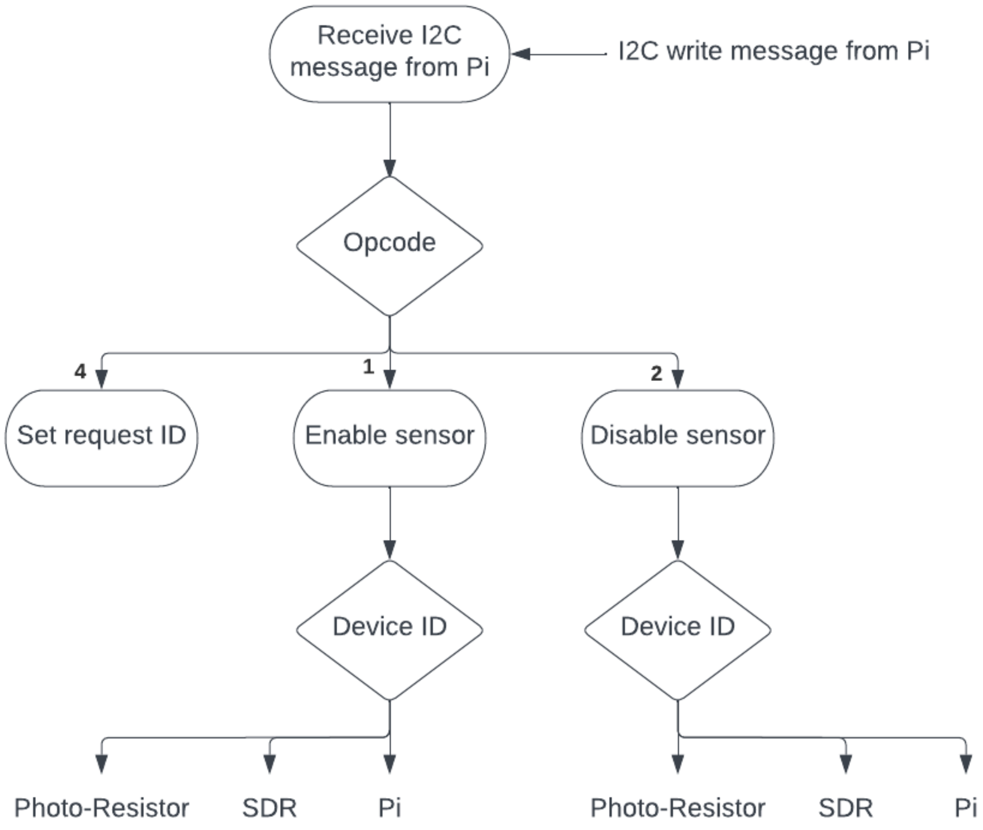


**Figure 17.** receiveEvent() function decision tree

The ESP32's receiveEvent() function calls the following functions depending on the opcode sent by the Pi:

- restart_device(int device_id)
- shutdown_device(int device_id)

The functionality to set the current_variable to be equal to the data sent is uncomplicated and therefore is done in the receiveEvent() function instead of a dedicated function.

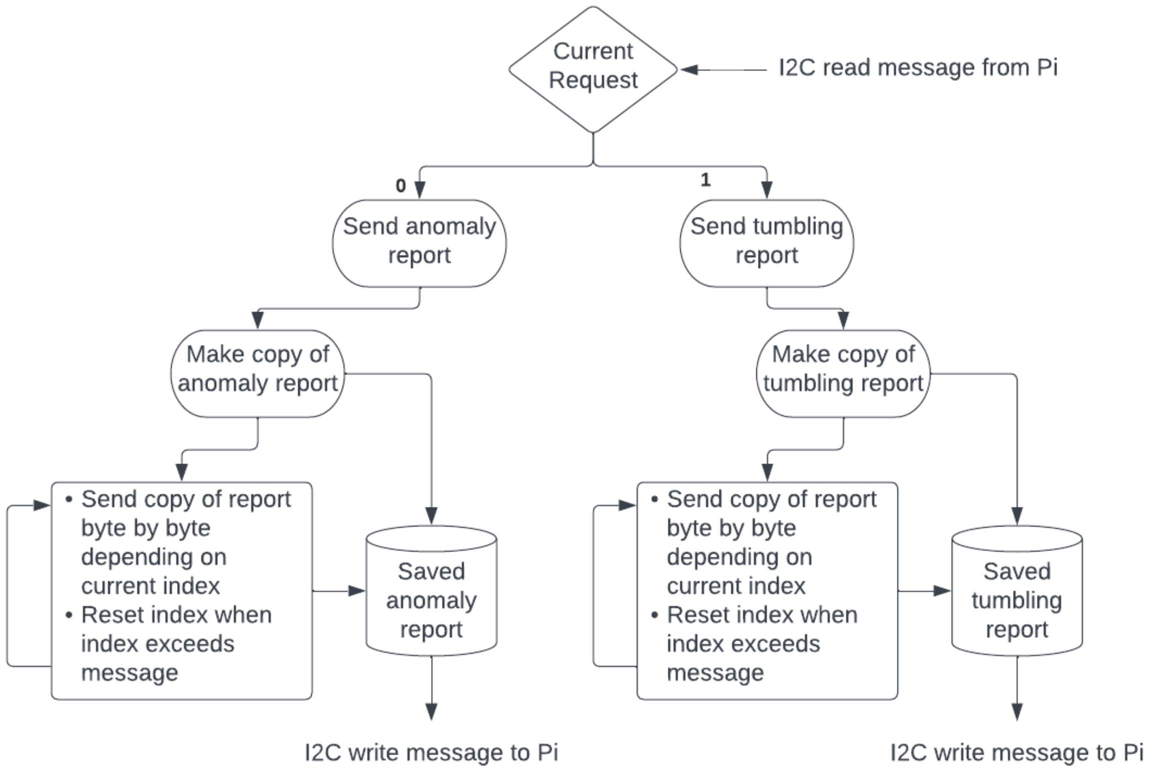


**Figure 18.** requestEvent() function decision tree

The ESP32's requestEvent() function calls the following response functions depending on the current request variable:

- send_anomaly_report()
- send_tumbling_report()
- send_current_sensor_readings()

While the send_current_sensor_readings() was implemented successfully when using the ESP32-S3 devkit, after the switch to the ESP32-MINI-1U we were unable to get this data transfer working. Fortunately, if the current sensor readings were outside of an acceptable range, this was reported in the anomaly report, so while some functionality was lost, the ability to transfer crucial information was retained.

The ESP32 uses the Wire.h library to handle I2C communications, and the Pi uses smbus2. The ESP32Client class is present in both the messages.py and onoff.py scripts in the Pi/ directory, and all of the ESP32 I2C functions are in the PiComms.cpp file in the ESP32/PiComms/ directory. See Appendix B to view selections of the software and links to the complete software listing.

## *3.6 Interfaces*

The ESP32's I2C bus was designed to communicate with the IMUs and the fuel gauges, but the fuel gauges' SDA and SCL lines ended up swapped on the final board design. We were able to establish functionality with the fuel gauge included in the subsystem integration demo, but cut the fuel gauge from the final demonstration due to this issue. The ESP32's I2C bus is the TwoWire ESPBus, and is sent to the Adafruit_ICM20948 icm instance used to read data from the IMU.

The ESP32 also reads in analog voltage input from the current sensors for the ESP32's IMU, the Pi's IMU, and the Pi itself in the measure_imu_currents() function in the Sensors.cpp file (see entry ii in Appendix B). The analog voltage is read in as a value between 0 and 4095, which maps to 0 to 3.3V on the input pin. Each pin is sampled 1,000 times before averaging the sensor value and completing the calculation of the current. The sensor value is multiplied by (3.3/4095.0) to calculate the voltage on the pin, and then the zero value is subtracted from the measurement. The zero value is the voltage read on the pin when the device is off and the current is known to be zero; the current sensors can measure both positive and negative voltage, so this is roughly half of 3.3V. This voltage value is then divided by the sensitivity; for example, the IMU's current sensors measure 0.132A for every 1V, so the IMU voltages are multiplied by 1/0.132. The Pi's current sensor operates at 5V, but is sent through a voltage divider before being read in from the ESP32, so this voltage also has to be multiplied by 3.3/5 in addition to being divided by the sensitivity in order to calculate the current through the Pi. A Pi current anomaly is triggered if the calculation of the current is greater than 1A or less than 100mA, and an IMU current anomaly is triggered if the voltage that corresponds to the current is more than 0.02V above or below the expected zero value. This is because current powering the IMUs are below the sensitivity of the current sensors.

## *3.7 Communications*

The communications subsystem consists of a Sidekiq Z2 SDR from Epiq Solutions, which we are responsible for powering and providing sensor data. The Sidekiq Z2 SDR was chosen because it was donated to us by Epiq Solutions, with a retail cost of roughly $20,000. It is also very small (30mm x 51mm x 5mm), very light (8 g), relatively low power (<2W), wideband (70 MHz to 6 GHz), has a high sample rate (61.44 Msps), and is able to interface with Raspberry Pi.

The SDR is powered by a 9V DC signal from a 2.5*0.7mm barrel plug connector. This barrel plug connector connects to the output of the 9V MOSFET network through a through hole connector, with and SDR enable signal from the ESP32 as the voltage read on the gate of the N-Channel MOSFET at the beginning of the network, as discussed in section 4.4. Since the Proto-Sat team has not yet developed the software interface between the SDR and their Raspberry Pi flight computer, for now, the data packetization requirement of this subsystem is fulfilled by sending battery level and detected anomalies notifications to the flight computer over an I2C connection between the ESP32 and the Pi. In the future, this data will be the input to signal processing software that will modulate, encode, and send the data to the SDR for transmission.

**Figure 19**. Sidekiq Z2 software-defined-radio

## 4 System Integration Testing

### *4.1 Subsystem Testing*

Since almost the whole system was contained entirely on the board, the board was built first before any testing could be done. This involved placing all surface mount components, although through-hole components were left unconnected for the time being. In practice, due to difficulties in placing the ESP32, an ESP32-S3 Devkit module was used for initial testing by soldering wires from the Devkit's GPIO pins to the appropriate places on the board.

Using a power supply, about 3.7V was supplied to the BT1 connector, which provided power to the whole board save for the 9V supply. After it was determined that the RPM5.0-2.0 would not be able to output 5V and the decision was made to use two batteries in series, the test voltage was moved up to 6V. With this, the 1.8, 3.3, and 5V converters were tested with multimeters. After the 9V converter board was connected, the power supply was moved to the BT2 connector with a 3.7V supply and the 9V lines were tested with a multimeter as well.

To test charging, batteries were connected to the BT1 & BT2 connectors. A green LED connected to the PGOOD pins of the MPPTs ensured that the batteries were recognized as valid input sources. Then, solar panels were connected to VSC1 & VSC2. When a large light was shone on them, the red LEDs connected to the CH pins lit up, indicating that the MPPT was charging the batteries using the solar panels.

To ensure proper testing of the anomaly detection system, software needed to be written. “Hello, World!” code was written to be installed on the ESP32 using PlatformIO, Visual Studio Code, and an Arduino framework, so that the ESP32’s connections and download boot could be tested. Then, GPIO pins corresponding to enable signals for different devices were toggled periodically, and a multimeter was used to ensure the power lines of each device were actually changing. Additionally, in this stage, code was developed for the current sensors, and the currents were displayed on the serial monitor while the devices toggled. The difference in readings indicated the current sensors were functioning properly, and the current sensors were subsequently calibrated to each device's respective zero and typical operating values.

The devices tested in this way included the IMUs, the Pi Zero, and the SDR. For testing purposes here and elsewhere, the Pi was connected using Dupont wires. Moreover, while the Pi was on, we used the ssh protocol to connect to it. Its ability to ssh indicated that it was working as expected.

To test the photoresistor and its circuit, we wrote high the GPIO32 pin, which provided its power. Then, the analog signal of the photoresistor was monitored using the serial monitor while the photoresistor was exposed and covered. A drastic difference in values between these states indicated the circuit was working properly.

Once the devices could be turned on, I2C lines were tested. Both the ESP32’s bus with the fuel gauges and IMU and the Pi’s bus with the ESP32 and another IMU were tested. Unfortunately, we found the fuel gauges unable to be used due to an error in schematic design. Signals were read from the IMUs using an Adafruit library for the ICM-20948, and signals were traded between the Pi and ESP32 using standard I2C code. To ensure proper functioning of the IMUs, the board was shaken and rotated to indicate tumbling while the serial monitor was observed.

After the IMUs were confirmed to be working, the state detection system was tested in full by creating the environmental conditions needed to toggle between each state. On boot, the board was tested to ensure that only the IMU would be on. The board was then shaken to simulate tumbling, then the photoresistor was alternately covered and uncovered to simulate day and night cycles.

## *4.2 Testing Outcomes*

By taking measurements from the outputs of the converters, we guarantee our power system's basic functionality. The charging requirements are also guaranteed by observing the MPPTs and their LEDs. By testing the ESP32's ability to enable and disable devices, the basic capability of one main feature of the anomaly and state detection subsystem is shown. Another capability of this subsystem is shown by the current sensor, photoresistor, and 9V analog readings as the finite state machine progresses between states. This subsystem is essentially completed by demonstrating these two processes in conjunction.

The tumbling detection is shown by the ESP32 receiving proper signals from the IMUs when tumbling is being simulated.

The Interfaces subsystem is demonstrated with the proper functioning of the I2C busses between all devices; particularly, the testing demonstrated that data could be transferred from the ESP32 and read on the Pi via the ssh protocol.

Since all devices that were connected properly were able to function properly under expected conditions (tumbling, light, no light, charging, battery-powered, etc.), we also concluded that noise in the circuit was not pronounced enough to cause interference, although more robust testing may be needed for a final design. Based on measurements taken with a multimeter, we also concluded that the system was properly grounded, with continuity between ground pins, and no devices were left floating when not being powered. The size limitations were considered satisfied when the fully-built board was able to be placed in the ProtoSat without extra adjustment to the ProtoSat's design.

## **5 Users Manual/Installation manual**

## *5.1 Installation*

If not already, the board must be constructed by placing components and ICs. It is recommended to use a pick-and-place machine and a reflow oven for this process. The pick-and-place should be used in conjunction with a schematic of the complete board design, as the board itself does not have labeled values. Leave off the through-hole components and the PMOSes (Q2, Q3, Q5, Q7, Q9, Q10, Q11); NMOSes may be placed normally. In accordance with the ESP32-MINI-1U datasheet, ensure the reflow process heats the board above 217°C (~235-250°C) for 60-90 seconds.

Then, place the through-hole components. Connect the Pi Zero 2 W and the external 9V converter board using header pins. Connect BT1, BT2, and the solar cell inputs to connectors compatible with the desired batteries and solar cells. BT1 should be connected to a rechargeable battery providing at least 5V, while BT2 should be connected to one providing at least 3.7V. Connect the photoresistor, ensuring adequate length of wires so that the photoresistor can stick outside the CubeSat. Finally, connect header pins to the ESP32 programming contacts so that a programmer board can be used.

The configuration of the MOSFETs deviates from the original board design for proper functioning. During board design, the source and drain for each PMOS configuration were flipped, so the PMOSes must be turned upside down and rotated in order to contact the correct pads. Consult the datasheet and the schematic to ensure that the source, drain, and gate are placed correctly. Additionally, two wires and a resistor (~10 kΩ) must be added. **Figure 20** below shows how these circuits should be modified. Specifically:

- the input pins of each IMUs' current sensor (VDD_IMUPI_IN & VDD_IMUESP_IN) should be shorted to 1.8V (VCC) with a wire;
- a 10 kΩ resistor should connect +5V & N$15;
- a PMOS with a low to mid-range threshold voltage (<1.0V, i.e. FDN352AP) should be placed so that the source is connected to +5V, the gate to N$15 (controlled by PI_EN), and the drain to POWER_PI_IN, the input of the Pi's current sensor;
- a PMOS with a mid-range threshold voltage (1.0-2.5V, i.e. FDN352AP) should be placed so that the source is connected V9V, the gate to N$4 (controlled by SDR_EN), and the drain to SDR, the voltage that the SDR receives.

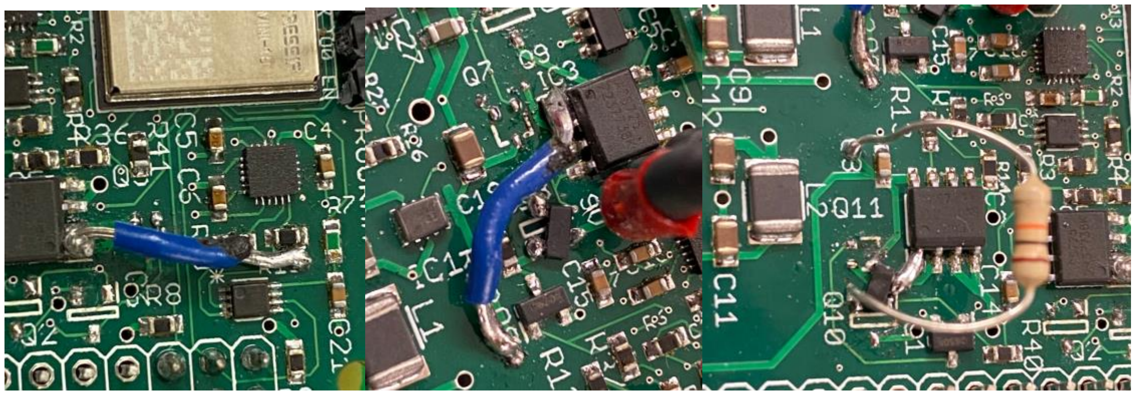


**Figure 20.** Modifications made to the board to ensure proper MOSFET circuit configurations.

Once built, the main board and its peripherals may be placed inside the ProtoSat. Place the board inside the Sat, and use the 2.4mm screw holes on the corners of the board for mounting. Once code has been written into the ESP32 (below), remove the programming pins and ensure the ESP32 starts in flash boot when connected to power.

## 5.2 *Setup*

Download the code from GitHub or from the senior design website. Create a PlatformIO project with VSCode, and select the ESP32 Dev Module option. Copy the PiComm/, Sensors/, and Tumbling/ directories from the ESP/ directory into the lib/directory of the project, and replace the main.cpp and platform.ini files in the base directory with the main.cpp and platform.ini files in the ESP/ directory. Add the following Adafruit libraries to the project: Adafruit Unified Sensor, Adafruit ICM20X, and Adafruit MAX1704X. Compile and upload the code to the ESP32-MINI with an ESP32 programmer. The ESP32 programmer can be connected via header pins next to the ESP32-MINI on the board.

Connect to the SDNet wifi in the Senior Design Lab. Once you see the green LED on the Pi on, run the command ssh pi@192.168.10.140 to ssh into the Pi Zero. Note that it will take a while for the Pi to boot, even after the green LED turns on. If you are unsure if the Pi is up and running, run ping 192.168.10.140 to see if the data packets sent are timing out. If they are timing out, try restarting the Pi. If none of the above works, run dns-sd -G v4 hal.local. If an entry is returned, check the IP address of your ssh command and the entry matches. If not, retry the ssh command with the newly found IP address. Once you have ssh-ed into the Pi, run the command cd SeniorDesign to enter the SeniorDesign demonstration directory, and run python messages.py or python onoff.py to run the demonstration scripts. The messages.py script will repeatedly ask the ESP32 for updates on anomalies and tumbling state, and requires no user input. The onoff.py script can be used to turn devices on and off depending on user entry; enter on:5 to the prompt to turn the SDR on. The two devices available for turning on and off are the Pi with device ID 4 and the SDR with device ID 5, although, if you enter off:4, the Pi will be powered off and you will no longer have a working SSH connection to the Pi.

## 5.3 *Signs of a Working Product*

Upon powering through batteries, the green LEDs connected to the MPPTs, PG1 & PG2, should light up, indicating valid input sources. Upon connecting solar panels and shining lights on them, the red LEDs CH1 & CH2 should light up. The ESP32 itself should be able to program. Once “tumbled” (i.e. shaken or rotated), the Pi Zero should also boot, indicated by a green LED (note: for the demo code, the Pi Zero is on automatically). Proper functioning of the Pi Zero should be ensured by using the ssh protocol; this may require light shining on the solar panels for a proper amount of current draw. The Pi’s programs may be run to ensure correct operation; see above for more details.

## *5.4 Troubleshooting*

There are myriad ways to troubleshoot the board, and which method should be used is based on the specific issue being met. In general, any troubleshooting may start with a visual inspection of the board. Ensure there are no pins shorted and that there are good electrical connections between components and the board. Look especially at contacts between the board and externals. Loose copper or excessive solder may create shorts around the board.

For power issues, connect BT1 and/or BT2 to a power supply, depending upon which device is misbehaving (e.g. for the 9V converter and SDR, connect to BT2; for all other devices, connect to BT1). Provide a voltage of at least 5V to BT1 or 3.7V to BT2. Check that all outputs of the converters are at expected values; the 1.8V converter and 9V converter may need to be enabled by the ESP32. If there is a discrepancy between expected and actual voltage outputs, check for poor connections from the specific converter, and consider replacing it.

For issues involving ESP32 programming, ensure that the pins IO0 and EN are being controlled by the programmer. If not, consider replacing the programmer, or using wires to connect these pins to an appropriate value. Ensure that the correct framework is being used on the IDE (i.e. an ESP32 devkit framework, not an ESP32-C3 or an ESP32-S3 framework). If issues still persist, consider replacing the ESP32 or building a new board.

For issues involving I2C connections, ensure that the bus is declared correctly in the code. The ESP32 should be a master on the bus with pins GPIO33 & GPIO25 respectively, and a slave with pins GPIO22 & GPIO19 respectively. Consider checking these pins with a logic analyzer configured to read I2C signals. If issues still persist, test code may be written to ensure the GPIO pins may be pulled high and low. For the IMUs, Ensure that the level shifters are working properly; if the GPIO pins are pulled high to 3.3V, the corresponding SDA & SCL pins on the level shifter that are connected to the IMU should be pulled to 1.8V. If the GPIO pins are pulled to ground, those pins should be pulled to ground as well. For the Pi, try ssh’ing and seeing if code can be run. If this does not work, try rebooting the system with a higher voltage (do not exceed 8.4V) or with solar cells with lights shining attached.

To troubleshoot the photoresistor, while it is receiving power, monitor the output voltage using a serial monitor or a multimeter. Note the reading when the photoresistor is exposed to ambient light. Then, totally cover the photoresistor in darkness by clasping it with both hands. The resistance should change, which would be reflected by a change in voltage; if not, consider replacing the photoresistor.

## 6 To-Market Design Changes

Quite a few changes will need to be made to our board before it will be ready to be placed in our CubeSat and launched into orbit. To begin, the board needs to be changed to account for some of the issues that were discussed above. Specifically, changes will need to be made to the MOSFET circuit system in order to ensure proper operation; this edit will not be hard to do but will require reprinting the boards. In addition, we want to add another MOSFET circuit for ProtoSat’s motor controller board. As we have it right now, the motor control board connects directly to the 9 volt converter, while the SDR connects to 9 volts through our board. For a future iteration, we will add a header pinout to attach the motor controller board to and create a MOSFET network that is similar to the ones we have for other components. This MOSFET network will ensure that the motor controller board can be turned off and disconnected from our flight computer without us losing communication with the SDR.

We will need to find a new 5 volt converter to use on our final board for our satellite. We had issues with the one that we chose as it seemed to only be a buck converter as opposed to a buck-boost converter. As a result, the converter could not boost the 3.7 volt to 4.4 volt input signal to 5 volts, but instead needed a signal that was greater than 5 volts. To make the converter work for our demo, we connected two batteries in series to produce a greater voltage on the 5 volt converter. We cannot use this design on our CubeSat for two specific reasons. The first reason is that we do not want to have a third battery due to mass and space constraints. The second and more important reason is that we would be unable to charge two batteries that are connected in series, so the only battery that could get charged would be the one supplying power to the 9 volt rail. As a result, it is obvious that we need to find a replacement part for the 5 volt converter for our final product.

We also will need to edit the fuel gauges, as we accidentally flipped the SDA and SCL pins in our schematic. We are also considering changing the fuel gauges as the ones we chose do not have great documentation. We neglected to include a current sensor for the SDR on our original design due to time, budget, and space constraints. Since the SDR is such a functional part of our satellite and to detect anomalies on all of our sensors and interfaces, we would definitely add a current sensor for the SDR before calling this a final product. Due to the high voltage (9V) and max current draw during transmission (300 mA), we would need to find a current sensor that is properly rated, which might be hard to find and costly.

In addition, we will replace the current sensors that are included in our design with different current sensors. We originally thought that we would have to read larger currents than we actually needed to. Due to this belief, we chose current sensors that had high current tolerances but were not very sensitive to smaller changes in current. Upon testing, we discovered that we actually need to measure very small currents (in the microamp to milliamp to range) that can also handle reading larger spikes in current in case of an anomaly. Thus, we will look deeper into finding a more suitable current sensor for future iterations of our board. We are currently considering either the ACS724LLCTR-2P5AB-S, which has a sensing range of ±2.5A, or the ACS724LLCTR-05AU-S, which has a sensing range of 0-5A. Both of these sensors have a sensitivity of 800mV/A, which would be reduced to about 528mV/A when used with the 3.3V pins of the ESP32. Alternatively, other options such as low input bias current op-amps could be explored.

While placing everything on the board, we also came across a number of different ways to improve the layout of the board. For example, the position of the header pin spaces for the Pi can be shifted to another portion of the board and we can place less components with significant height around it in order to fit the Pi on top of the board as opposed to on the bottom. Shifting parts around will be made significantly easier if we also had bottom mounted components. We thought about utilizing bottom mounted components when making our original board design, but we could not due to potential concerns from the ProtoSat team in regards to the position of the board in the CubeSat. More research can be done on if bottom mounted components will be possible to use on our board if the ProtoSat team deems it necessary. We also put too many vias to ground under the 5 volt and 3.3 volt converters, which are both parts that get mounted to the board from the bottom. If we had misplaced these parts by just a few millimeters, one of the pads could have made contact with the ground via and it would not have worked. Thus, we should decrease the amount of ground vias for the converters and move them out from under the components.

The biggest design consideration that will need to be made is if the components we chose are suitable to launch into space. Space hardened materials and other components that are rated for space are not cheap, not easy to come by, and are normally made to order, so the lead time can be months. Due to the cost and time restraints, we could not make our board completely ready to launch. Instead, we made the board to fit the prototype CubeSat that was being made by the ProtoSat team. When our club gets selected for a launch, the team will need to build an entirely new system that is prepared for launch, and our board will be included in this redesign.

## 7 Conclusions

To conclude, we deem our project as a success for the club. Although our board is not a final product that can be put into a satellite and launched, we knew coming into this project that this would be the case due to some of the aforementioned reasons (i.e.

cost and time). Choosing to complete this project allowed us to consolidate almost all of our electronics into a compact series of boards that fit within the dimensions given by the ProtoSat team, thus making our club more attractive in the long run to be selected for a CubeSat mission.

## 8 Appendices

### *8.1 Appendix A. Complete Hardware Schematics*

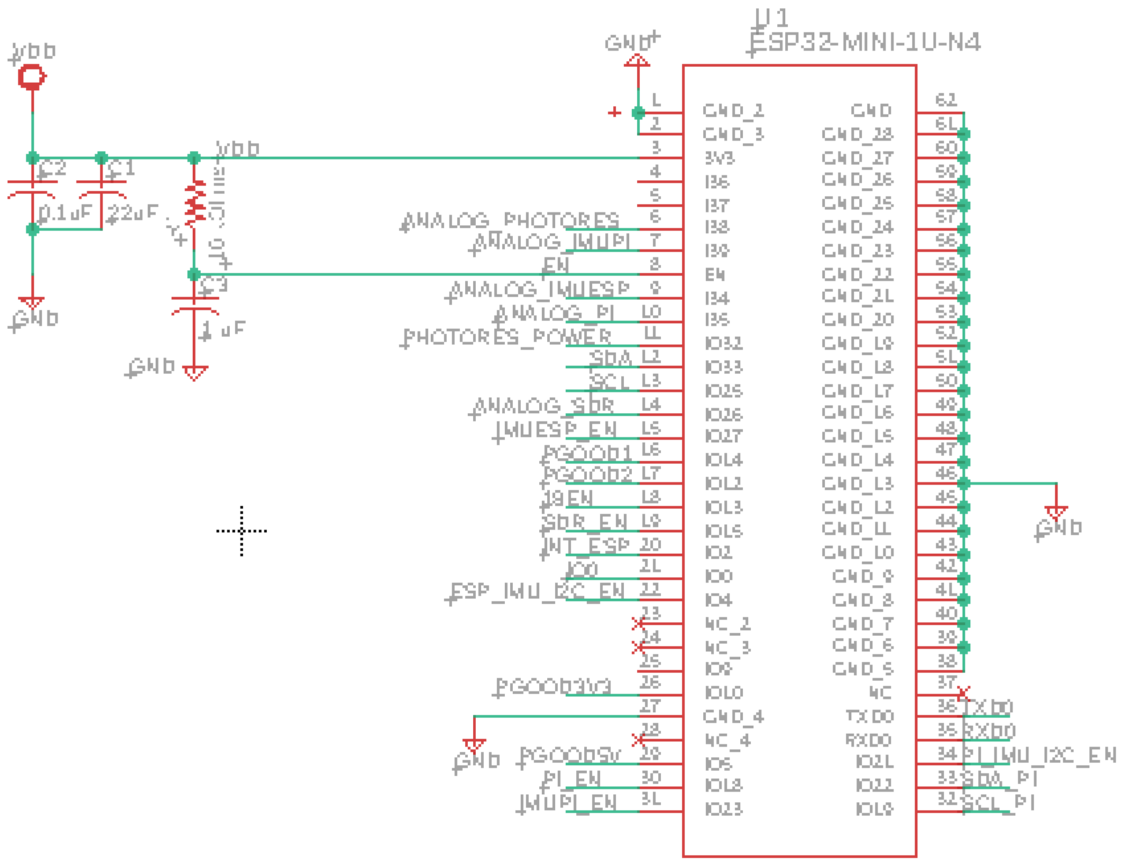


**Figure 21.** ESP32 Schematic

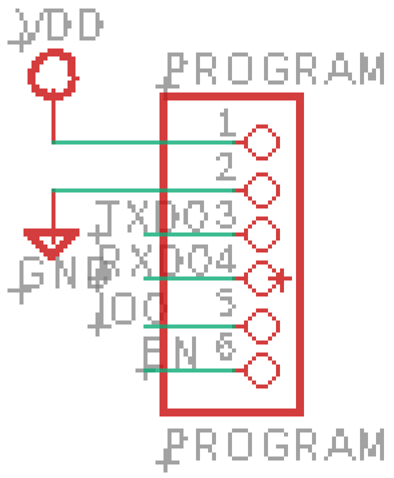


**Figure 22.** ESP32 Programming Pins

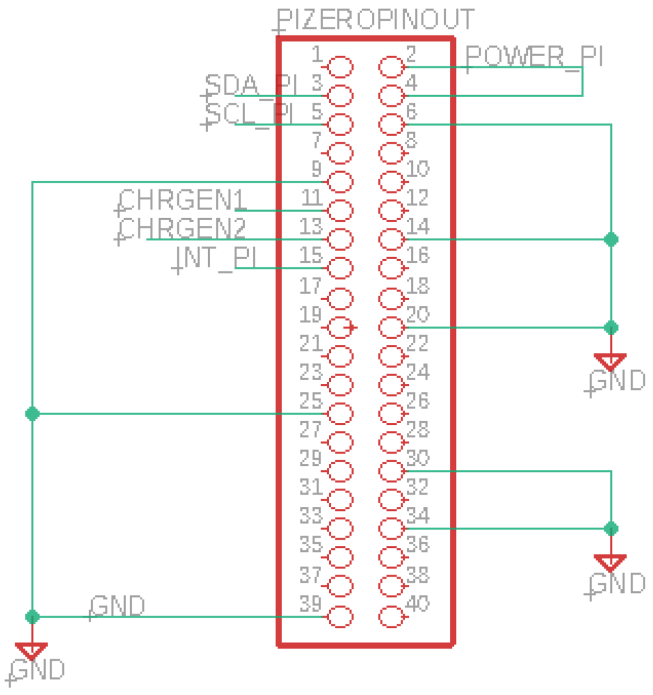


**Figure 23.** Pi Zero Schematic

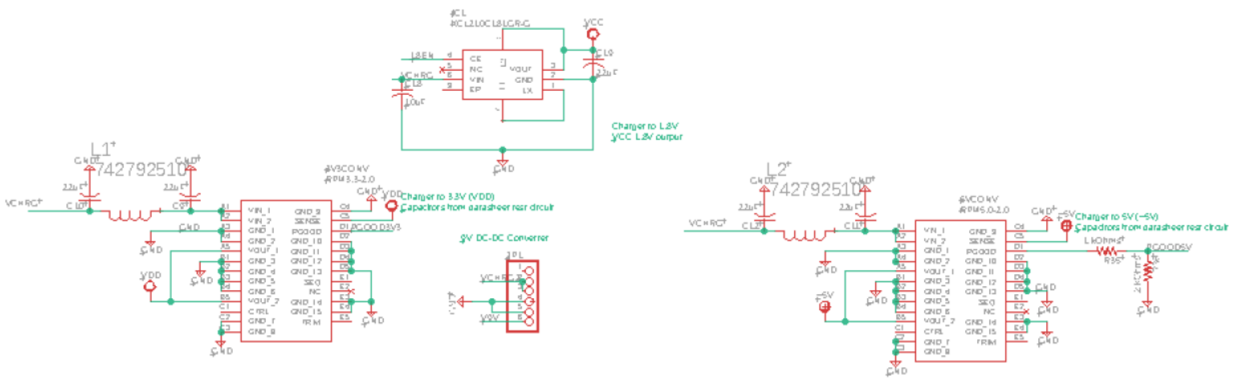


**Figure 24.** 1.8V, 3.3V, 5V, and 9V Converter Schematics

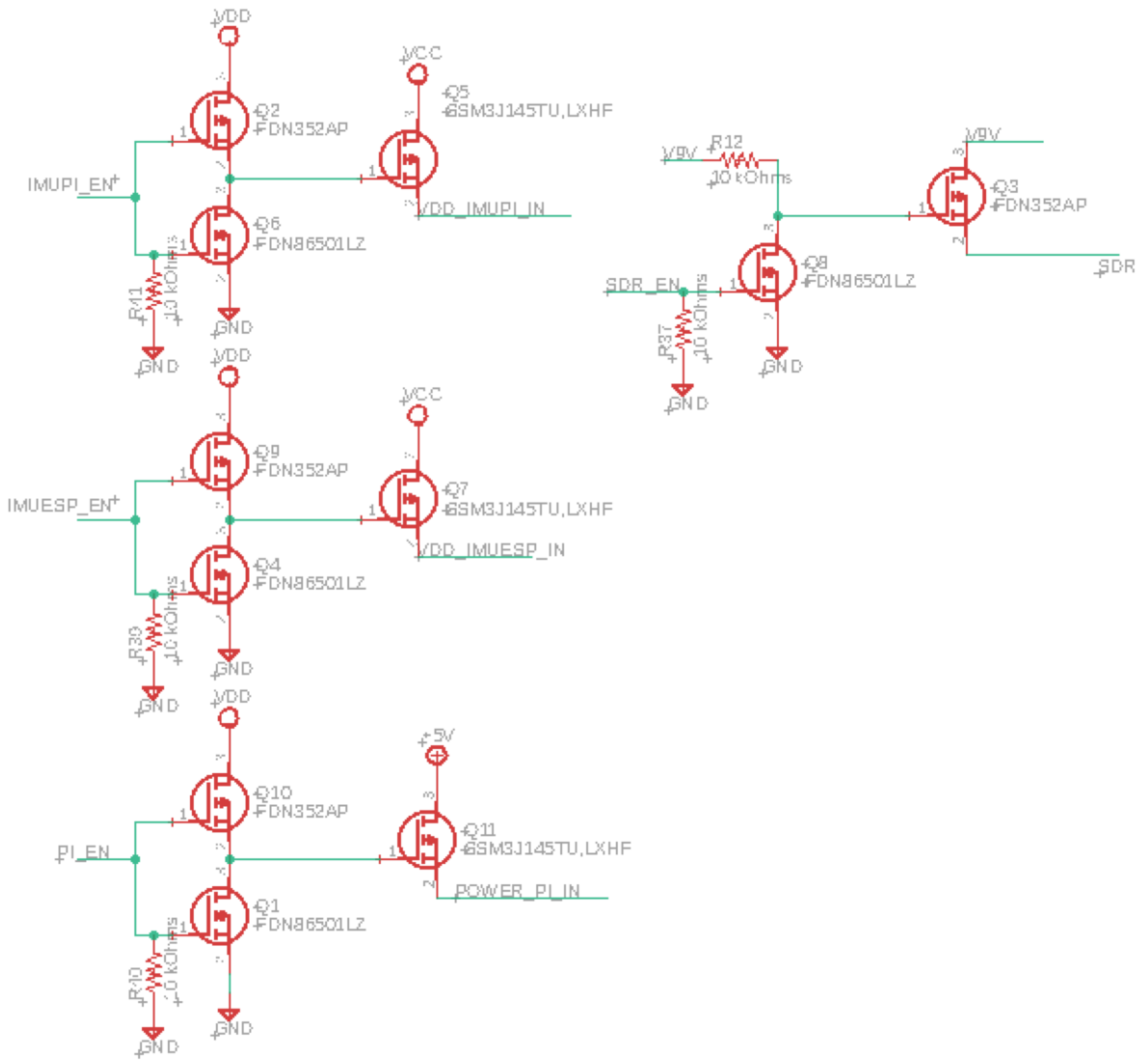


**Figure 25.** MOSFET Circuit Schematics

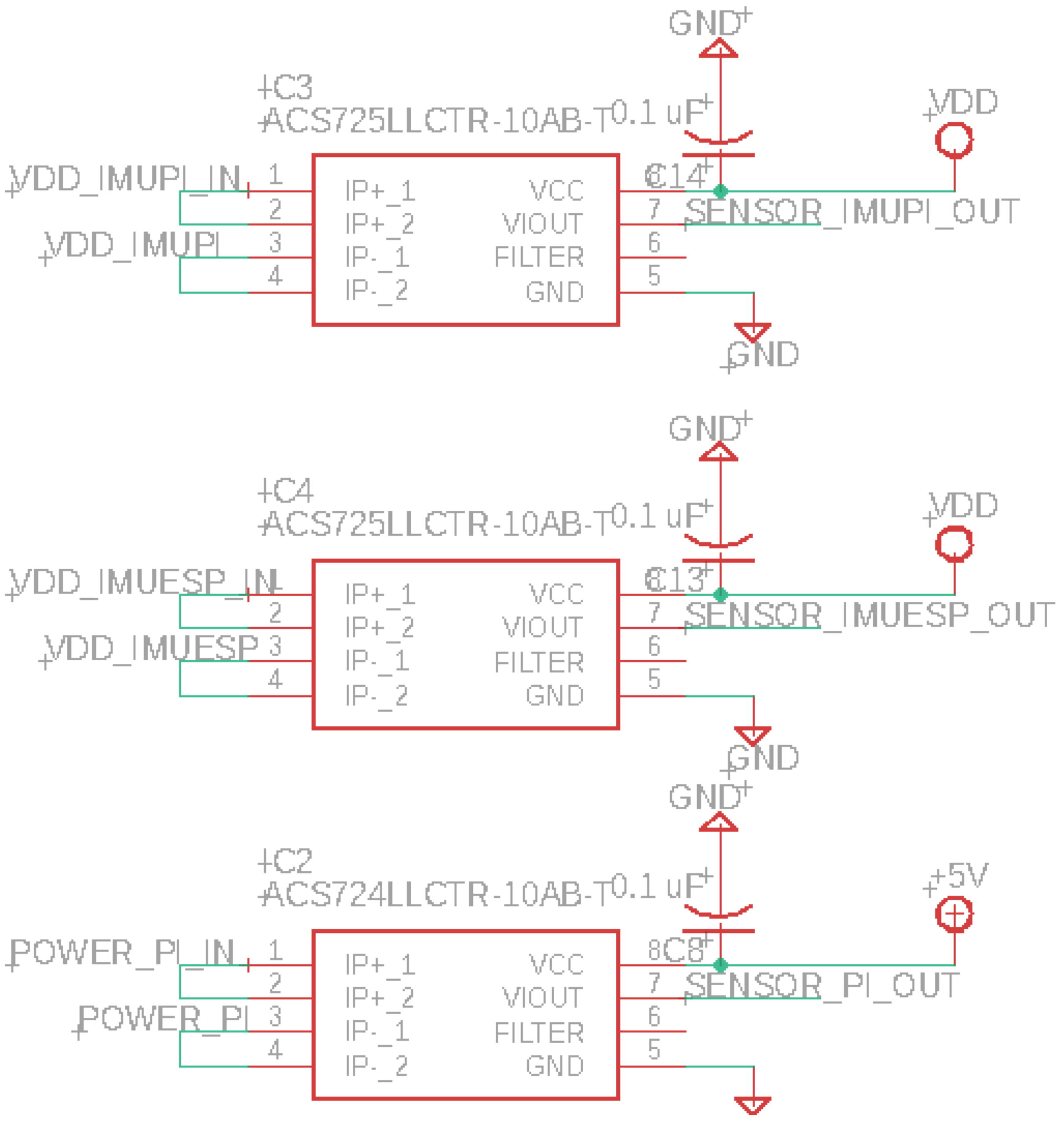


**Figure 26.** Current Sensor Schematics

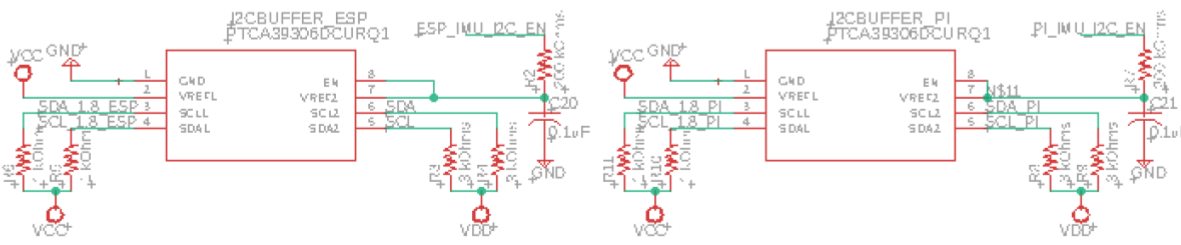


**Figure 27.** Level Shifter Schematics

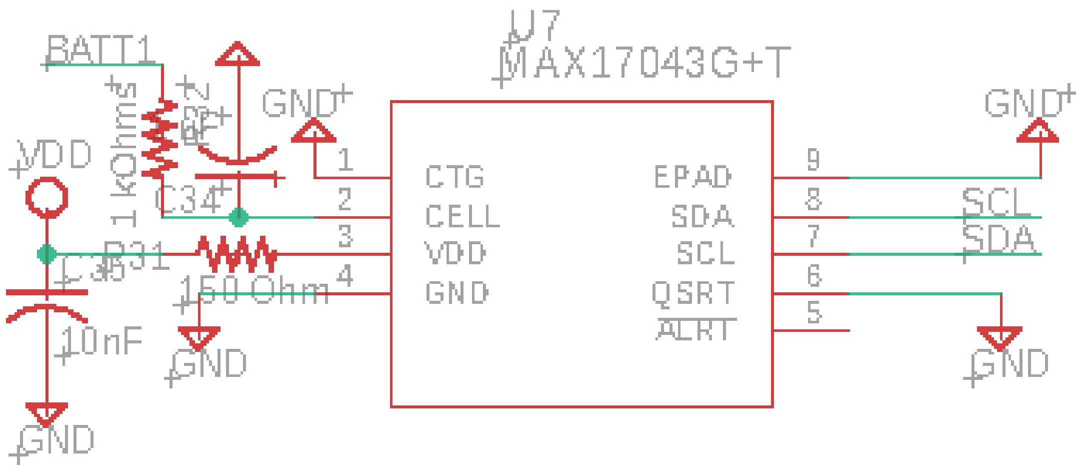


**Figure 28.** Fuel Gauge Schematic

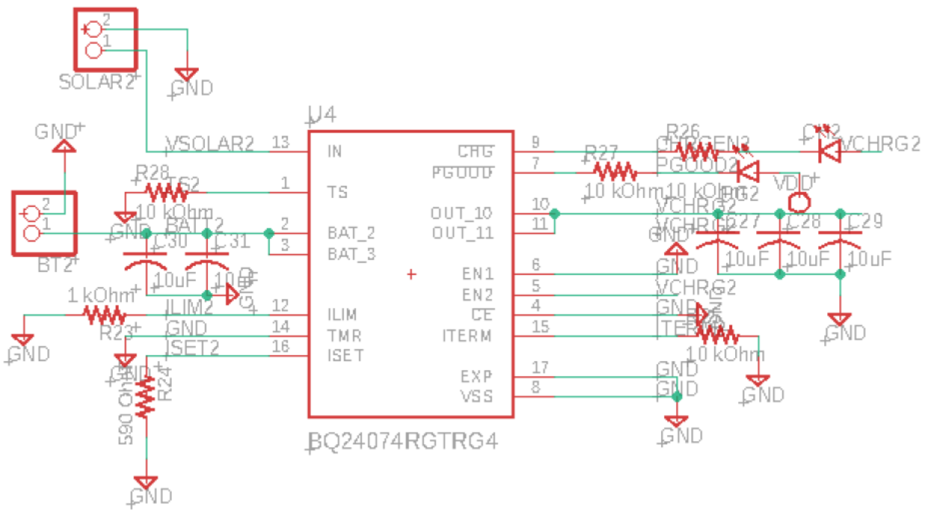


**Figure 29.** MPPT Linear Charger

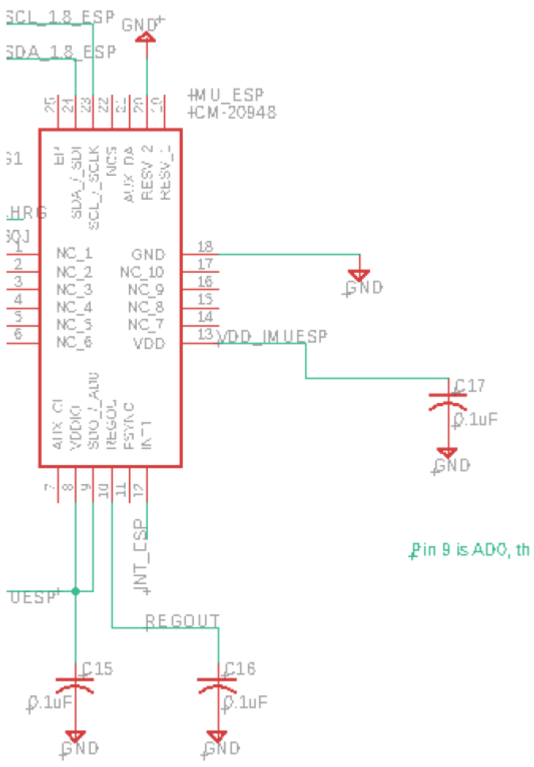


**Figure 30.** IMU Schematic

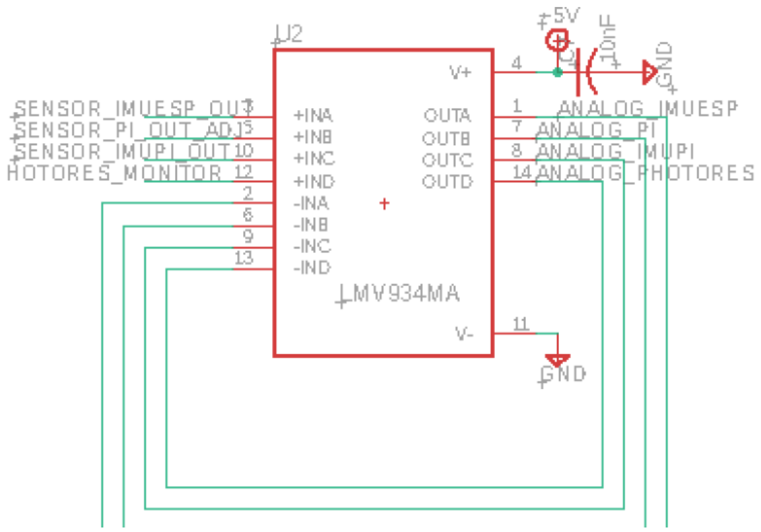


**Figure 31.** Unity Gain Op Amp Schematic - Quad Input

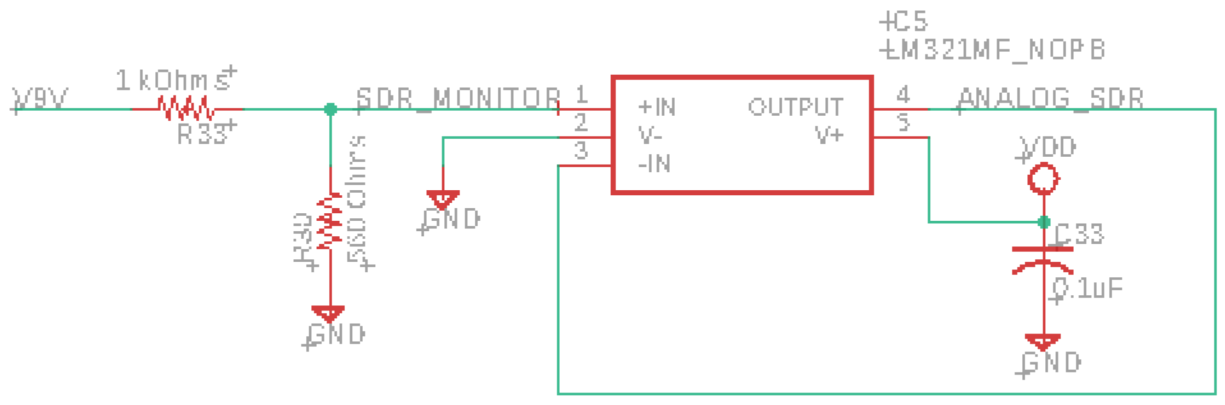


**Figure 32.** Unity Gain Op Amp Schematic - Single Input

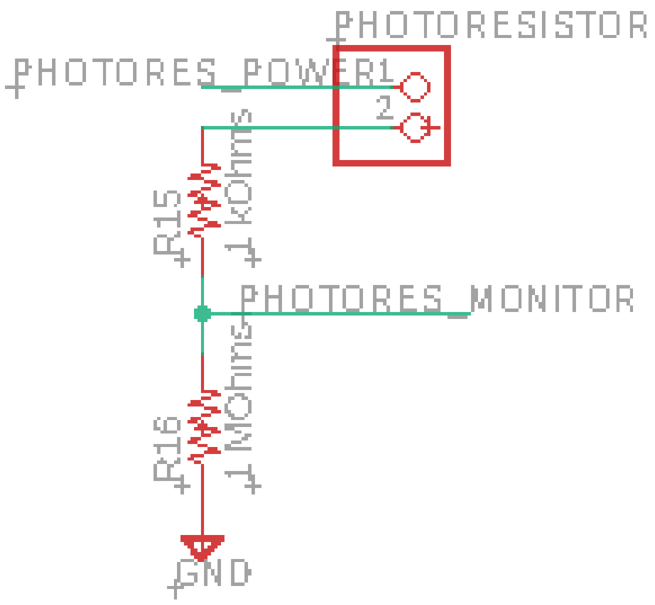


**Figure 33.** Photoresistor Circuit Schematic

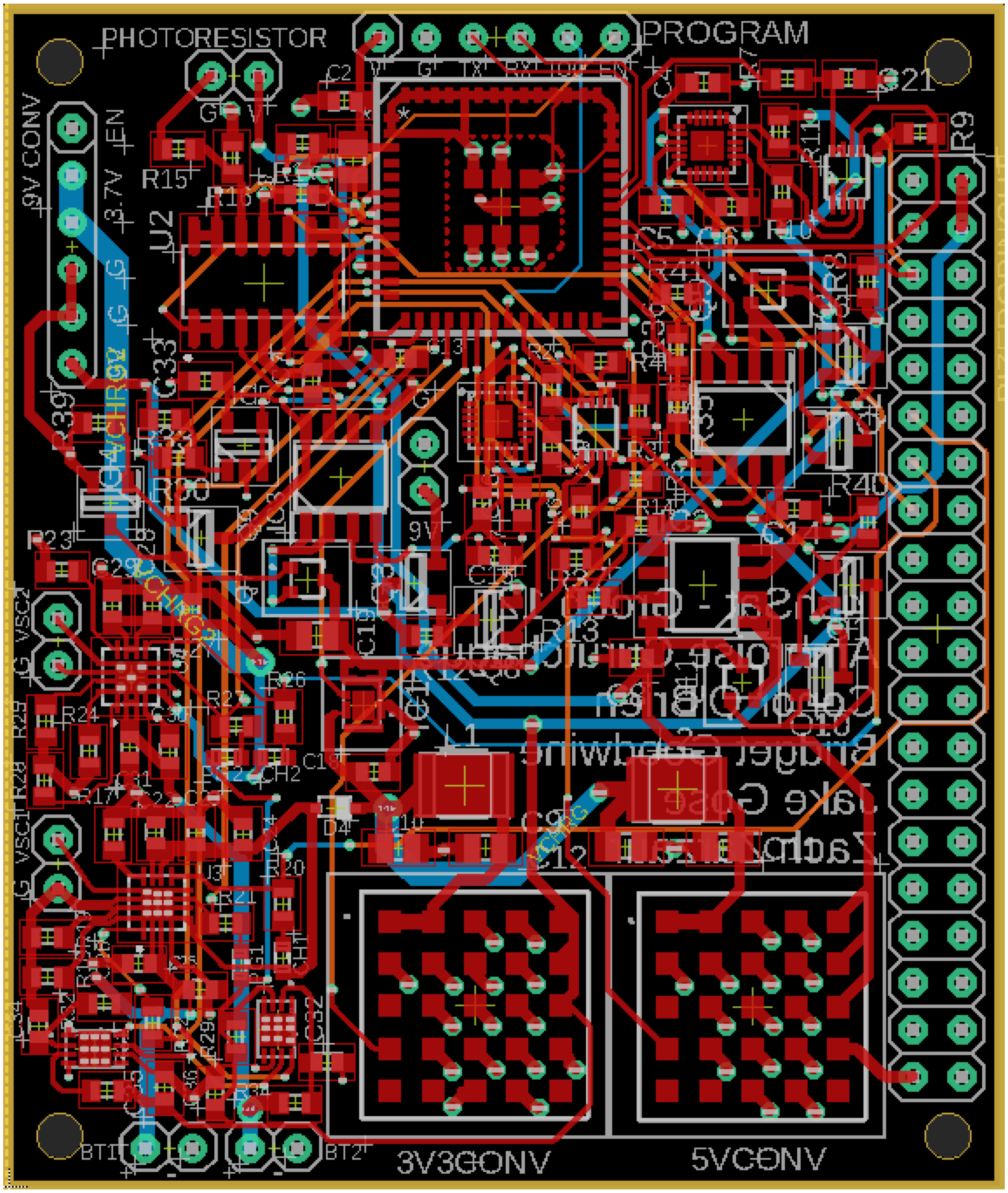


**Figure 34.** Final Board Layout

## 8.2 *Appendix B. Complete Software Listings*

i. main.cpp for the ESP32 (main.cpp)

```
// Include the Wire library for I2C
#include <Wire.h>
#include "Arduino.h"
#include <Adafruit_Sensor.h>

// Private libaries
#include "PiComms.h"
#include "Tumbling.h"
#include "Sensors.h"

// External TwoWire I2C wires from Tumbling.h and PiComms.h
extern TwoWire PiBus;
extern TwoWire ESPBus;
extern Adafruit_ICM20948 icm;
extern Adafruit_MAX17048 maxlipo;

// global tumbling/anomaly values
extern String current_anomaly_report;
extern String current_tumbling_report;
extern int current_tumbling_state;

// global current values
extern double piIMUCurrent;
extern double piCurrent;
extern double espIMUCurrent;

// Tumbling external global state variables
extern int currentState;
extern int oldCurrentState;
extern int anomaly;
extern float outputsArray[3];
extern float tumbleTime;
extern float tumbleStart;
extern float stillStart;
extern float stillTime;
extern float batteryLevel;
extern float oldBatteryLevel;
extern float battThreshold;

void turnOnPi() {
  digitalWrite(Pi5_en, HIGH);
  digitalWrite(PI_IMU_en, HIGH);
  digitalWrite(LVL_SHFT_PI_EN, HIGH);
  Serial.println("Pi, Pi IMU, and Pi level shifter enabled.");
  delay(1000);
  PiBus.begin((uint8_t)ESP_ADDR, (int)PI_SDA, (int)PI_SCL);
  Serial.println("Joined Pi I2C bus.");
}

void turnOffPi() {
  // turn off pi
  // Pi low, IMU pi low, pi imu level shifter, SCL/SDA lines of
pi bus
  digitalWrite(Pi5_en, LOW);
  digitalWrite(PI_IMU_en, LOW);
  digitalWrite(LVL_SHFT_PI_EN, LOW);
  Serial.println("Pi, Pi IMU, and Pi level shifter disabled.");
}

void setup() {
  // Serial monitor initialization
  Serial.begin(115200);
  delay(500);
  while(!Serial);

  // Register PiComms.h functions to handle I2C data from Pi
  PiBus.onReceive(receiveEvent);
  PiBus.onRequest(requestEvent);

  pinMode(PR_en, OUTPUT);           // devices enable pins
  pinMode(Pi5_en, OUTPUT);
  pinMode(SDR_en, OUTPUT);
  pinMode(IMU_en, OUTPUT);          // IMUs
  pinMode(PI_IMU_en, OUTPUT);

  pinMode(LVL_SHFT_EN, OUTPUT);     // level shifters
  pinMode(LVL_SHFT_PI_EN, OUTPUT);
  pinMode(DCDC_EN, OUTPUT);         // DC-DC converter

  // enable i2c communication periferals
  digitalWrite(LVL_SHFT_EN, HIGH);
  digitalWrite(LVL_SHFT_PI_EN, HIGH);
  Serial.println("Level shifters for Pi and ESP's IMU enabled.");
  digitalWrite(DCDC_EN, HIGH);
  Serial.println("DC-DC converter enabled.");

  delay(10);

  // Begin IMU I2C connection on ESP controlled bus
  ESPBus.begin((int)ESP_SDA, (int)ESP_SCL);
  Serial.println("Joined ESP's I2C bus.");
  // Initialize current sensor for analog input & current
measurement
  pinMode(PI_IMU_CURRENT, INPUT);
  pinMode(ESP_IMU_CURRENT, INPUT);
  pinMode(PI_CURRENT, INPUT);

  // Write low all devices except IMU to initialize
  digitalWrite(PR_en, LOW);
  Serial.println("Photoresistor: OFF");
  digitalWrite(IMU_en, HIGH);
  Serial.println("IMU: ON");
  digitalWrite(SDR_en, LOW);
  Serial.println("SDR: OFF");

  // turn off pi
  turnOnPi();
  delay(1000);

  // Initialize MAX1704X (fuel gauge)
  /*if(!maxlipo.begin(&ESPBus)) {
    Serial.println("Oops, no MAX17048 detected...");
  } else {
    Serial.print(F("Found MAX17048"));
    Serial.print(F(" with Chip ID: 0x"));
    Serial.println(maxlipo.getChipID(), HEX);
  }*/

  // Initialise IMU
  Serial.println("Adafruit ICM20948 test!");
  if (!icm.begin_I2C((uint8_t)0x69, &ESPBus, (int32_t)0)) {
    Serial.println("Failed to find ICM20948 chip");
  } else {
    Serial.println("\nSUCCESS!\n");
  }

  Serial.print("Gyro range set to: ");
  switch (icm.getGyroRange()) {
  case ICM20948_GYRO_RANGE_250_DPS:
    Serial.println("250 degrees/s");
    break;
  case ICM20948_GYRO_RANGE_500_DPS:
    Serial.println("500 degrees/s");
    break;
  case ICM20948_GYRO_RANGE_1000_DPS:
    Serial.println("1000 degrees/s");
    break;
  case ICM20948_GYRO_RANGE_2000_DPS:
    Serial.println("2000 degrees/s");
    break;
  }

  uint8_t gyro_divisor = icm.getGyroRateDivisor();
  float gyro_rate = 1100 / (1.0 + gyro_divisor);

  Serial.print("Gyro data rate divisor set to: ");
  Serial.println(gyro_divisor);
  Serial.print("Gyro data rate (Hz) is approximately: ");
  Serial.println(gyro_rate);



  Serial.println("I2C connections ready.");
  Serial.println("Device Control System Initialized.");
  Serial.println("Anomaly/State Detection System Initialized.");
  Serial.println("Undeployed state detected...");

}

void loop() {
  delay(1000);
  static int i = 0;

  oldBatteryLevel = batteryLevel;

  // retrieve current anomaly report
  currentState = checkState(currentState, oldCurrentState,
batteryLevel);
  updateState(currentState, oldCurrentState);
  anomaly = checkAnomalies(currentState);
  batteryLevel = 0;
  String old_anomaly_report = sendReport(anomaly,
oldBatteryLevel, batteryLevel);
  //Serial.print("Old anomaly report: ");
  //Serial.println(old_anomaly_report);

  oldCurrentState = currentState;

  // retrieve current data as anomaly report
  current_anomaly_report = measure_imu_currents(piIMUCurrent,
espIMUCurrent, piCurrent);
  current_anomaly_report = "  " + current_anomaly_report;
  Serial.println("Current anomaly report: ");
```

```
    Serial.println(current_anomaly_report);

    // save current cycle state for tumbling report
    current_tumbling_state = currentState;
    Serial.print("Current detected state: ");
    Serial.println(current_tumbling_state);

  }
```

ii. Links to GitHub for the libraries written for the ESP32 main.cpp program

- PiComm.cpp
- PiComm.h
- Sensors.cpp
- Sensors.h
- Tumbling.cpp
- Tumbling.h

iii. platform.ini configuration file

```
; PlatformIO Project Configuration File
;
;   Build options: build flags, source filter
;   Upload options: custom upload port, speed and extra flags
;   Library options: dependencies, extra library storages
;   Advanced options: extra scripting
;
; Please visit documentation for the other options and examples
; https://docs.platformio.org/page/projectconf.html

[env:esp32dev]
platform = espressif32
board = esp32dev
framework = arduino
monitor_speed = 115200
lib_deps =
     adafruit/Adafruit Unified Sensor@^1.1.9
     adafruit/Adafruit ICM20X@^2.0.5
     adafruit/Adafruit MAX1704X@^1.0.0
```

iii. ESP32Client class for the Pi

```
#!/usr/bin/env python

# SDA1 GPIO2 pin 3
# SCL1 GPIO3 pin 5

#    J8
# 1     2
# SDA   5V
# SCL   GND

# connect GPIO2 on Pi to 22 on ESP32 (SDA)
# connect GPIO3 on Pi to 21 on ESP32 (SDL)

# ls /dev/*i2c* results in /dev/i2c-1

import array
import struct
import time
from smbus2 import SMBus

class ESP32Client:
    def __init__(self):
        # initialize i2c parameters
        self.addr = 0x8 # ESP32 address
        self.offset = 0
        self.bus = SMBus(1) # 1 = /dev/ic2-1
        self.num_current_bytes = 8 # size of a double

    # send one 32B message to Pi
    def send_msg(self, msg, verbose=False):
        if len(msg) > 32:
            print('ERROR: msg is > 32B, use send_long_msg >:(')
            return False

        # normalize to 32B
        while len(msg) < 32:
            msg = msg + '\n'

        if verbose:
            print(f'Sending 32B message: {msg.strip()}')
            print(f'Length of message:   {len(msg)}')
            print('')

        # convert to byte array
        byte_msg = bytearray()
        byte_msg.extend(map(ord, msg))
        if verbose:
            print(f'bytes: {byte_msg}')
            print(f'len:   {len(byte_msg)}')
```

```
                print('')

            # write bytes
            self.bus.write_i2c_block_data(self.addr, self.offset,
    byte_msg)
            if verbose:
                print('Done sending.')
            return True

        # request status report from the ESP32
        def get_anomaly_report(self):
            # send status report request to esp
            self.send_msg('4:0')
            msg = '0:' + '\0'
            if not self.send_msg(msg):
                return ''

            msg = ''
            # read response from esp
            byte = self.bus.read_byte(self.addr)
            msg += chr(byte)
            #print(f'Skipping: {byte} -> {hex(byte)} -> {chr(byte)}')
            time.sleep(0.25)
            byte = self.bus.read_byte(self.addr)
            msg += chr(byte)
            #print(f'Skipping: {byte} -> {hex(byte)} -> {chr(byte)}')
            time.sleep(0.25)
            while byte != 0:
                byte = self.bus.read_byte(self.addr)
                msg += chr(byte)
                #print(f'{byte} -> {hex(byte)} -> {chr(byte)}')
                time.sleep(0.25)
            msg += chr(self.bus.read_byte(self.addr))
            time.sleep(0.1)

            # remove opcode:
            #msg = msg[2:]

            return msg

        def get_tumbling_report(self):
            # send status report request to esp
            self.send_msg('4:3')
            msg = '0:' + '\0'
            if not self.send_msg(msg):
                return ''
            msg = ''
            # read response from esp
            #msg += chr(self.bus.read_byte(self.addr))
            byte = self.bus.read_byte(self.addr)
            letter = chr(byte)
            #print(f'Skipping: {byte} -> {hex(byte)} -> {chr(byte)}')
            time.sleep(0.25)
            byte = self.bus.read_byte(self.addr)
            letter = chr(byte)
            #print(f'Skipping: {byte} -> {hex(byte)} -> {chr(byte)}')
            time.sleep(0.25)
            #msg += letter
            while byte != 0:
                byte = self.bus.read_byte(self.addr)
                letter = chr(byte)
                #print(f'{byte} -> {hex(byte)} -> {chr(byte)}')
                msg += letter
                time.sleep(0.25)
            msg += chr(self.bus.read_byte(self.addr))
            time.sleep(0.1)

            # remove opcode:
            #msg = msg[2:]
            return msg


        # request message from ESP32 for SDR
        def get_msg(self):
            # opcode 4 sets request id, request id 0 is send message
            self.send_msg('4:2')

            # read in 8 bytes = sizeof(double)
            msg = ''
            for i in range(0, 23):
                msg += chr(self.bus.read_byte(self.addr))
                time.sleep(0.05)
            time.sleep(0.1)

            print(msg)
            return msg

        # request current data from ESP32
        def get_current_sensor_readings(self):
            # opcode 4 sets request id, request id 1 is send sensor
    readings
```

```
        self.send_msg('4:1')

        # read in 8 bytes = sizeof(double) for Pi current
        status = bytearray()
        byte = self.bus.read_byte(self.addr)
        #print(f'Skipping: {byte} -> {hex(byte)} -> {chr(byte)}')
        time.sleep(0.25)
        for i in range(0, 8):
            #status.append(self.bus.read_byte(self.addr))
            byte = self.bus.read_byte(self.addr)
            status.append(byte)
            #print(f'{byte} -> {hex(byte)} -> {chr(byte)}')
            time.sleep(0.25)
        time.sleep(2)
        piIMUCurrent = struct.unpack('d', status)[0]

        # read in ESP's IMU's current
        status = bytearray()
        byte = self.bus.read_byte(self.addr)
        #print(f'Skipping:{byte} -> {hex(byte)} -> {chr(byte)}')
        time.sleep(0.25)
        for i in range(0, 8): # 8 bytes = sizeof(double)
            #status.append(self.bus.read_byte(self.addr))
            byte = self.bus.read_byte(self.addr)
            #print(byte)
            status.append(byte)
            #print(f'{byte} -> {hex(byte)} -> {chr(byte)}')
            time.sleep(0.25)
        time.sleep(2)
        espIMUCurrent = struct.unpack('d', status)[0]

        # read in Pi's IMU's current
        status = bytearray()
        byte = self.bus.read_byte(self.addr)
        #print(f'Skipping: {byte} -> {hex(byte)} -> {chr(byte)}')
        time.sleep(0.25)
        for i in range(0, 8):
            #status.append(self.bus.read_byte(self.addr))
            byte = self.bus.read_byte(self.addr)
            #print(byte)
            status.append(byte)
            #print(f'{byte} -> {hex(byte)} -> {chr(byte)}')
            time.sleep(0.25)
        time.sleep(2)
        piCurrent = struct.unpack('d', status)[0]
        # read in battery &
        status = bytearray()
        byte = self.bus.read_byte(self.addr)
        #print(f'Skipping: {byte} -> {hex(byte)} -> {chr(byte)}')
        time.sleep(0.25)
        for i in range(0, 8):
            status.append(self.bus.read_byte(self.addr))
            #print(f'{byte} -> {hex(byte)} -> {chr(byte)}')
            time.sleep(0.25)
        time.sleep(2)
        batteryPerc = struct.unpack('d', status)[0]

        return piIMUCurrent, espIMUCurrent, piCurrent,
batteryPerc

    def restart_sensor(self, sensor_id):
        print(f'Sending request to restart sensor {sensor_id}')
        msg = '1:' + str(sensor_id) + '\0'
        self.send_msg(msg)

    def shutdown_sensor(self, sensor_id):
        print(f'Sending request to shutdown sensor {sensor_id}')
        msg = '2:' + str(sensor_id) + '\0'
        self.send_msg(msg)

    def misc_msg(self, msg):
        print(f'Sending misc. message to ESP32: {msg}')
        opcode = '3'
        msg = '3:' + msg + '\0'
        self.send_msg(msg)
```

iv. Scripts utilizing this class for the demonstration

- messages.py
- onoff.py

## 8.3 *Appendix C. Relevant Parts and Component Datasheets*

Most recent BOM - please see for all capacitors and resistors used

*List of Relevant Parts (Mouser/Digikey page with datasheet is linked)*

- 1.8V DC-DC Buck Converter
- 3.3V DC-DC Buck Converter
- 5V DC-DC Boost Converter

- 9V DC-DC Boost Converter
- Ferrite Bead
- Raspberry Pi Zero 2W
- IMU
- ESP32-MINI-1U-H4 Module
- Level Shifter
- Battery Fuel Gauge
- Schottky Diode
- Unity Gain Op Amp - Quad input
- 3.3V Current Sensor
- 5V Current Sensor
- Unity Gain Op Amp - Single input
- pMOS
- nMOS
- MPPT Controller
- Photoresistor